RESOURCE MANAGEMENT FOR DATA STREAMING APPLICATIONS

A Thesis
Presented to
The Academic Faculty

by

Bikash Kumar Agarwalla

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August, 2010

RESOURCE MANAGEMENT FOR DATA STREAMING APPLICATIONS

Approved by:

Dr. Umakishore Ramachandran,
Committee Chair
College of Computing
Georgia Institute of Technology

Dr. Umakishore Ramachandran,
Advisor
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Ling Liu
College of Computing
Georgia Institute of Technology

Dr. Ann Chervenak
Center for Grid Technologies
USC Information Sciences Institute

Dr. Brian Cooper
Senior Research Scientist
Yahoo! Research

Date Approved: June 30, 2010

ACKNOWLEDGEMENTS

I deeply appreciate the committee members who evaluated this dissertation and provided me valuable guidance along the way. Their feedbacks and guidance has helped me mature as a researcher. I sincerely thank Karsten Schwan, Ann Chervenak, Ling Liu, and Brian Cooper for being part of my committee. Ann deserves a special mention for hosting me for a week at USC Information Sciences Institute. During my visit at USC, I got hands-on experience on grid technologies and it solidified the foundations of this dissertation.

During my stay at Georgia Tech, I have been fortunate enough to come across and collaborate with many smart and knowledgeable colleagues, many of whom I consider my close personal friends. Sameer Adhikari and Arnab Paul have made me feel welcome in Atlanta as seniors and helped me mature as a graduate student. Rajnish Kumar, Matthew Wolenetz, Junsuk Shin, Hasnain Mandviwala, Nova Ahmed, David Hilley, Dave Lillethun, Marthin Modahl, Nissim Hareel, Namgeun Jeong, Xiang Song, and Sandip Agrawal have been both friends and mentors to me helping me solve the right problems. Prahlad Fogla, Aranyak Mehta, Parikshit Gopalan and Tejas Iyer are old friends from my undergraduate study at IIT Bombay and I have been fortunate enough to carry on some parts of my graduate student journey together with them at Georgia Tech.

I have encountered long list of passionate faculty and staff during my career at Georgia Tech. Mustaque Ahamad and H. Venkateswaran deserve special mention for having motivated me achieve excellence in a technical career. Among the many staff members I relied on, Barbara Binder, Dani Denton, Cathy Dunnahoo, Susie McClain, and Deborah Mitchell deserve special mention for helping me promptly with countless

administrative tasks. I also want to thank Alan Glass, Ellen Zegura, and Leo Mark for being constant source of support and for promptly helping me even when I was not physically present in school. I will fondly remember the guidance and inspirations from research scientist Phillip Hutto. I would also like to thank Matthew Wolf for the ease with which he made the computing infrastructure available.

My colleagues at HP Labs, Raj Kumar, Sujoy Basu, and Vanish Talwar, have helped me mature as a researcher in a corporate environment. I remember the wonderful times I had during my eleven months internship and I have been fortunate enough to get in-depth exposure to Grid Computing research during this period. My internship experience led the foundations for this work. I must also thank my colleagues at Facebook for helping me succeed as an Engineer.

It has been an honor for me to work with Kishore Ramachandran as an advisor. He has been the perfect guide, mentor, colleague, friend, and a source of inspiration. I can not thank him enough for helping me navigate the rough waters of research and academia. Always helping me look at the positive side and helping me find innovative solutions to the problems, Kishore is an absolute role model of an advisor.

I am also deeply appreciative of my extended family members. My uncle Omprakash Agarwalla, and my cousin Ajay Agarwalla deserve special mention for always motivating me to exceed expectations and to be a role model for my nephews and nieces.

I must dedicate this thesis to my family. I would like to express my deepest thanks to my wife, Shipra Agarwalla, for sacrificing so much, for being a constant source of inspiration and for motivating me to finish my dissertation. Without her support, I would be completely lost. I would like to thank my parents (Suresh Agarwalla and Muni Devi), my brother (Binit Agarwalla), and my sister (Ria Agarwalla) for always looking up to me as a role model and inspiring me to dream big and aim high. Without their love and support, this work would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This dissertation investigates novel middleware mechanisms for building streaming applications. Developing streaming applications is a challenging task because (*i*) they are continuous in nature; (*ii*) they require efficient transport of data from/to distributed sources and sinks; (*iii*) they need access to heterogeneous resources spanning sensor networks and high performance computing; and (*iv*) they are time critical in nature.

One common characteristics of these applications is data fusion. I present a novel programming abstraction, called DFuse, that makes it easier to develop fusion applications. The application program is specified as a dataflow graph with fusion points. DFuse middleware instantiates the graph on distributed resources and subsumes issues inherent in distributed programming - such as failures, partial fusion, buffer management, and synchronization. Through experiments, I demonstrate that DFuse API implementation has reasonable overhead.

I also address the challenges involved in allocating high performance computing resources for these applications. The scheduling framework consists of a heuristic algorithm, called Streamline, for placement of streaming application dataflow graph on HPC resources. I demonstrate the performance benefits of Streamline in a controlled environment through simulation as well as in wide area environment using Planetlab. Also I demonstrate that the scheduling algorithm can be implemented as a grid service and be deployed in wide area environment.

While Streamline does the placement for such streaming applications well, the application dynamics may result in the computation and communication characteristics of the application changing over time. I present a Distributed Scheduling heuristic

and a Periodic Streamline algorithm to address the limitations of Static Streamline algorithm. The performance of Distributed Algorithm is compared with Periodic Streamline and Static Streamline. Through micro measurements, I show that the Distributed Algorithm performs close to Periodic Streamline and 6x better than Static Streamline under dynamic resource availability. Through scalability study, I also show that the Distributed Algorithm performs close (within 5%) to Periodic Streamline algorithm with much less (7.5x less) overhead.

Finally, using a case study of such data streaming and ubiquitous application and the experience gained via building it, we propose a taxonomy of ubiquitous computing stack called UbiqStack. UbiqStack consists of five orthogonal functionalities of most commonly occurring subsystems for ubiquitous applications. Through the lens of the UbiqStack taxonomy, we survey a variety of subsystems designed to be the building blocks from which sophisticated infrastructures for ubiquitous computing can be assembled.

In summary, I develop Fusion Channel programming abstraction that makes it easier for domain experts to build data streaming applications. An application only needs to specify the input and output connections to fusion channels, and the fusion functions. The subsystems developed in this dissertation take care of instantiating an application, allocating resources for the application (via scheduling heuristics) and dynamically managing the resources (via dynamic scheduling). Through performance evaluation, I demonstrate that the resources are allocated efficiently to optimize the throughput and latency constraints of an application. Through extensive micro measurements and scalability studies, I have established my thesis: "An intuitive programming abstraction will make it easier to build dynamic, distributed, and ubiquitous data streaming applications. Moreover, such an abstraction will enable an efficient allocation of shared and heterogeneous computational resources thereby making it easier for domain experts to build these applications."

CHAPTER I

INTRODUCTION

Advances in sensing, computing, and communication infrastructure is paving the way for many demanding applications. Environment monitoring, distributed surveillance, disaster response, and traffic monitoring are all examples of such applications. These applications in their full form are capable of stressing the computing and communication infrastructures to their limit. For instance, in a real-time distributed surveillance application, the data may be captured through distributed cameras and be processed, aggregated in the sensor network before being communicated by designated sensor nodes to high performance computing (HPC) resources for heavy duty processing. These applications require access to resource continuum shown in Figure 1.

Streaming applications, as we refer to such applications, are typically represented as a coarse-grain dataflow service graph, wherein the nodes represent increasing sophistication of computational services that may need to be performed on the data stream to facilitate the extraction of high-level information. Consider for example, a video-based surveillance application. The compute intensive part may analyze multiple camera feeds from a region to extract higher level information such as "motion", "presence or absence of a particular face", or "presence or absence of any kind of suspicious activity". These applications have the following characteristics: (*i*) they are continuous in nature; (*ii*) they require efficient transport of data from/to distributed sources/sinks; (*iii*) they need access to resource continuum spanning sensor network and HPC resources; and (*iv*) they require efficient use of available resources to carry out compute-intensive tasks in a timely manner.

One interesting aspect of this emerging class of streaming applications is that they

are *ubiquitous* and *dynamic*. The application dynamism may result in the computation and communication characteristics of an application changing over time. Even the dataflow graph of the application could change depending on the result of the computation with the addition and deletion of new stages to the pipeline.

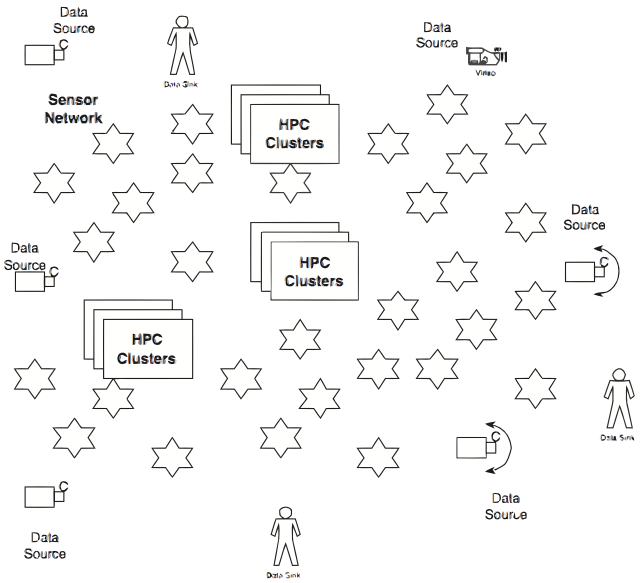


Figure 1: Resource Continuum

1.1 Problem Statement

As a case study of such data streaming and ubiquitous applications, I contributed to the development of a software intercom that allows users throughout a home to communicate through walls and floors as if they were in the same room. Using the experience gained during this system development and by analyzing existing technologies at the time, we came up with a taxonomy of ubiquitous computing software stack called UbiqStack [61]. UbiqStack consists of five orthogonal functionalities of most commonly occurring subsystems for ubiquitous applications. These include *Registration and Discovery*, *Service and Subscription*, *Data Storage and Streaming*, *Computation Sharing*, and *Context Management*. Out of these five functionalities, the focus of this dissertation is on Computation Sharing. Computation Sharing allows a

running application to make use of remote computational resources on demand. For example, a surveillance application running on a sensor network may decide to offload complicated vision tracking components onto more capable high performance computing resources (HPC) available in the ubiquitous computing infrastructure. However, facilitating Computation Sharing is a challenging task because of all the application characteristics mentioned previously. Moreover, distributed nature of these applications present additional issues (such as failure handling, buffer management, and synchronization) that are inherent in distributed programming.

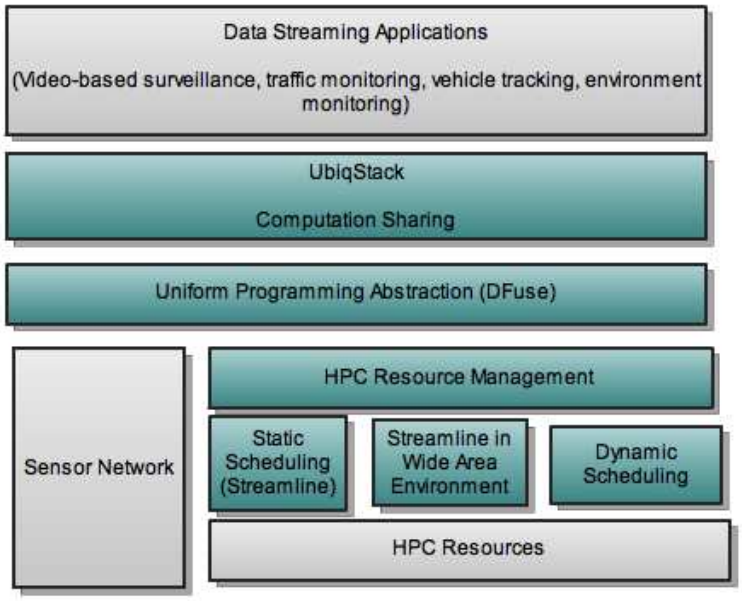


Figure 2: My Research Contributions (boxes highlighted in green)

My thesis is the following:

An intuitive programming abstraction will make it easier to build *dynamic*, *distributed*, and *ubiquitous* data streaming applications. Moreover, such an abstraction will enable an efficient allocation of *shared* and *heterogeneous* computational resources thereby making it easier for domain experts to build these applications.

Figure 2 shows the research contributions of this dissertation in support of the

thesis statement.

1.2 Contributions of this Dissertation

1. **DFuse Abstraction:** I identify the need for a uniform programming abstraction that hides the complexity of distributed programming and the underlying hardware while providing flexibility to develop various data streaming application. One common requirement for streaming application is data fusion, whereby, data from multiple input sources is *fused* together to derive higher level information. we present a uniform programming abstraction called DFuse[52, 69]. My contribution in DFuse is the design, implementation, and evaluation of the DFuse APIs. Using the DFuse APIs, application programmers need to only implement the fusion functions and provide the dataflow graph (the relationships of fusion functions to one another). The fusion API in the DFuse architecture subsumes issues such as data synchronization and buffer management. The API allows any synthesis operation on stream data to be specified as a fusion function, ranging from simple aggregation (such as min, max, sum, or concatenation) to more complex perception tasks (such as analyzing a sequence of video images). Through micro-benchmark of the primitives provided by the fusion API on an IPAQ farm, I demonstrate that DFuse API implementation has reasonable overhead. For operations on remote nodes, API overhead is less than 74.5% of the ideal cost. For operations with more than 20ms observed latency, API overhead is less than 53.8% of the ideal cost. API overhead is further reduced when we take into account various optimizations (such as pre-fetching, partial fusion, and data caching) that are implemented in DFuse. DFuse Implementation is used to evaluate various role assignment heuristics in sensor network. For example, we show that a distributed role assignment algorithm increases the application lifetime by 110% (by optimizing

power consumption) compared to static placement of the fusion functions.

2. **Streamline:** I design runtime mechanisms for the placement of streaming application dataflow graph on high performance computing resources. Grid computing offers the ability to harness the ambient HPC resources for a compute intensive problem. Most of the existing grid schedulers [19, 12, 79] focus on allocating resources for batch-oriented applications. While there have been efforts to expand the reach of the grid to support interactive [78] and streaming applications [21, 90], the resource management infrastructure available in the grid is largely geared to support batch-oriented applications. I develop a scheduling algorithm, called Streamline [8, 9], for allocating high performance computing resources for streaming applications. Streamline takes into account the following things in its scheduling decision: (a) computation and communication requirements of the various stages of the dataflow graph, (b) any application-specified constraints, and (c) current resource (processing and bandwidth) availability. The output of the scheduling heuristic is a placement of the stages of the pipeline on the available HPC resources such that the latency and throughput of the application are optimized. The performance of Streamline is compared with an Optimal placement, Simulated Annealing (SA) approximation to Optimal Placement, and E-Condor - a streaming grid scheduler I develop using Condor. Through extensive experiments, I show that Streamline performs close to Optimal and Simulated Annealing (within 1%), and is better than E-Condor by nearly an order of magnitude when there is non-uniform CPU resource availability, and by a factor of four when there is non-uniform bandwidth availability. Through scalability studies I show that Streamline is more effective than E-Condor in handling large dataflow graphs, and performs close to Simulated Annealing algorithms, with much smaller (by a factor of 1000) scheduling overhead.

3. **Scheduling in wide area environment:** In order to demonstrate the usefulness of Streamline scheduler in a wide area environment, I implement Streamline as a grid service in Globus Toolkit [35, 34] and evaluate the schedule generated by Streamline using Planetlab [68] resources. Planetlab provides a shared platform for distributed experiments, therefore, the resource availability is highly variable. I demonstrate that Streamline can be implemented as a grid service facilitating the allocation of resources and launch of streaming applications from a high level specification. Further, the experiments on Planetlab use real resource availability information by contacting various other grid services configured on each machine. The experimental results confirm that the schedule generated by Streamline is close to optimal placement (proxied by the Simulated Annealing algorithm) under real workload and in shared wide area environment.

4. **Dynamic Scheduling**: I address the dynamism inherent in scheduling data streaming applications. As demonstrated by experiments, Streamline does a good job for initial placement of streaming application dataflow graph on HPC resources. However, the dynamism in resource availability may result in streamline placement producing sub-optimal results over time. Therefore, I design and implement a periodic rescheduling algorithm using Streamline. I also present a Distributed Algorithm where each node of the dataflow graph independently makes its scheduling decision based on local information. The infrastructure consists of a programming primitive that facilitates dataflow graph migration, scheduling algorithm (Periodic Streamline and Distributed Algorithm), and architecture for failure handling. Through micro measurements, I show that the Distributed Algorithm performs close to Periodic Streamline and 6x better than Static Streamline under dynamic resource availability. Through scalability study, I also show that the Distributed Algorithm performs close to Periodic Streamline algorithm (within 5%) with much less (7.5x less) overhead.

5. **UbiqStack**: The programming abstractions and scheduling algorithms developed as part of this dissertation facilitate computation sharing and makes it easier to build data streaming applications. I also present all the components of UbiqStack that acts as a lens through which one can look at various systems for building ubiquitous data streaming applications.

Although I evaluate the scheduling algorithms in the context of Grid Computing, the system architecture and the algorithms are equally applicable in recent "Cloud Computing" model. Cloud Computing allows creation of customized virtual machine environments on top of physical infrastructure [11, 85]. The Cloud Providers allow the users to setup and customize the execution environment according to their application needs. Scheduling work in Cloud Computing [28] has focused on allocating virtual machines in a distributed and shared hosting environment provided by resource providers. My work is complementary and can be used by cloud providers to provision resources especially for data streaming applications.

The rest of this dissertation is organized as follows. In Chapter 2, I describe some example applications that motivate this work. In Chapter 3, I describe the DFuse programming abstraction. I also describe the key results from the role assignment heuristics for completeness. In chapter 4, I describe the Streamline heuristic for mapping streaming application dataflow graph to HPC resources. Chapter 5 describes my experience in building Streamline as a Grid service and using it in a wide area environment. Chapter 6 presents the system architecture and evaluation of the dynamic scheduling algorithm. Although this dissertation focuses on facilitating Computation Sharing, I describe all the components of UbiqStack in detail in chapter 7. Finally, I present conclusions in Chapter 8.

CHAPTER II

MOTIVATING APPLICATIONS

I present four representative applications to motivate this work. (*i*) Video based surveillance; (*ii*) A Real-time Traffic Monitoring application that requires continuous processing of traffic data; (*iii*) A Distributed Vehicle Tracking application that requires selective processing of data streams in order to track a suspicious vehicle in a timely manner; and (*iv*) A social computing application that provides recommendations based on location information derived from real-time audio and video analysis.

2.1 Video Based Surveillance

Video-based surveillance systems are used to monitor and identify "interesting events" under various scenarios. The events of interest depend on the target application for which surveillance is used. Consider for instance a threat identification and reduction scenario. In this scenario, cameras are deployed in a distributed fashion; the images from the cameras are filtered and fused together to extract higher level information such as "motion", "presence or absence of a human face", "presence or absence of any kind of suspicious activity". Security personnel can monitor these events and take appropriate action in real-time as soon as an event happens. With a large number of surveillance cameras, it becomes a more interesting issue. The key requirement for this application is that such an event should be discerned based on information collected from possibly thousands of cameras in real-time. Moreover, processing the video streams requires applying various compute-intensive vision processing algorithms on combination (fusion) of data streams. The real-time, compute-intensive, and distributed nature of the application make it quite challenging to build.

2.1.1 Real-time Traffic Monitoring

Monitoring, aggregating, and processing real-time traffic information has many uses including congestion control, route planning, trip time estimation and vehicle tracking. These applications, when integrated into the transportation systems infrastructure, have the potential to improve safety and enhance productivity [4]. With the advances in sensor technologies, it is now possible to capture the real-time traffic information 24 hours a day 7 days a week [3, 1]. Even the vehicles on road can serve as data collectors and anonymously transmit traffic and road condition information from every major road within the transportation network. Such information, when analyzed in real-time, can help in implementing active strategies to relieve traffic congestion. For instance, real-time traffic information can be used to find alternate routes in case of congestion or accidents and locate fuel efficient routes in order to save money.

Because of the large volume and time-critical nature of the data involved, the information needs to be processed in real-time. Moreover, the relevance of the data decays with time. Traffic conditions change very frequently and hence decision making, such as alternate route planning, depends on the most recent data. The compute intensive part of an alternate route planning application may consist of various services such as (*i*) image producer: for transporting data from distributed cameras to grid, (*ii*) background model: for creating a background model from the images, (*iii*) traffic density service: for estimating traffic density, and (*iv*) trip time prediction service: for predicting the trip time based on current traffic density and a map of the area. The compute intensive service graph of the application needs to be *mapped*, *composed*, and *managed* by a resource management system.

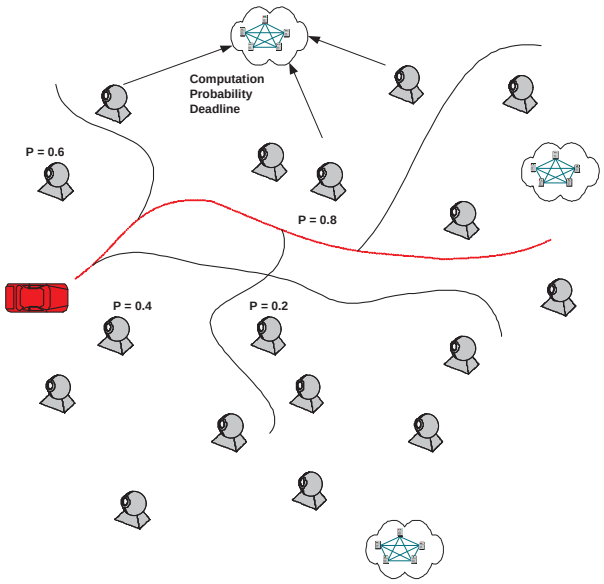


Figure 3: Vehicle Tracking Application

2.1.2 Distributed Vehicle Tracking

In this application, we consider the collaborative and selective use of data sources in order to track the trajectory of a particular vehicle. The system model consists of multiple cameras in the environment used to locate, track, and predict the trajectory of a suspicious vehicle. The application can assist in automatic tracking of suspicious vehicle, clearing traffic on the projected trajectory of the vehicle during high-speed chases thereby reducing traffic fatalities, and help with establishing road blocks at strategic points to bring the high speed chase to an early end. Figure 3 shows the system model for this application scenario. Each camera may launch computation on ambient HPC resources to predict the potential trajectory.

The prediction computation consists of a dataflow graph as shown in Figure 4. The camera are digitized, and processed by motion detector to identify moving vehicles; speed detector to identify speed of the vehicle; and target detector to discover a suspicious vehicle. The prediction module takes input from speed detector, target detector, and the present traffic condition in order to predict the trajectory of the vehicle. The prediction computation provides probabilities associated with different

trajectories a vehicle may take. These probabilities determine the relative importance of computation being launched on HPC resources by different data sources.

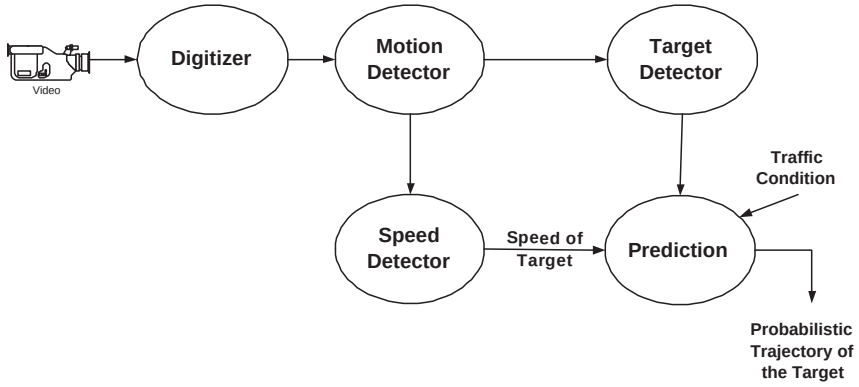


Figure 4: Dataflow Graph for Vehicle Tracking Application

The following characteristics distinguish this application: (*i*) Each computation has an associated deadline (time window) within which it must be completed. This deadline changes as recent data related to the vehicle is analyzed resulting in updated travel time estimates of the vehicle; (*ii*) Depending on the results of an earlier computation, later computations may be dropped; and (*iii*) each computation has different priority and the priorities change over time. Because of the large number of data sources involved, it is infeasible, often unnecessary, to process all data. Therefore, the resource management system should take into account the relative importance of the data and computation in its scheduling decisions.

2.1.3 Social Computing

Imagine a social service that can identify where your friends are and what they are doing based on audio and video feeds from multiple sources. Combining friends' location and activity data with their social graph contained in today's social networking sites such as Facebook lead to some very interesting applications. For instance, based on the analysis of video and audio streams coming from a person's social circle, a social computing service can notify you about your friends that are near your current

location without any active input from anyone. The service may get help to a friend in danger sooner by automatically identifying a threat scenario (as described in the first application). It can also help reduce money transfer scams that rely on scammer impersonating your friend and providing misleading information about their location. Some of these features can be supported by the current social networking sites with active location inputs from the users; however inferring context information based on audio and video analysis will provide additional value to the users.

Building such applications is challenging because it requires combining dynamic real-time data (from audio and video streams) with the slowly changing data represented by a person's social graph. In addition, the data volume that needs to be analyzed is huge. Facebook has 400M monthly active users now[2], and even if 0.1% of those users use such a service, it would mean analyzing 400K audio and video streams in real-time continuously.

2.1.4 Common Requirements

In this subsection, we identify the common requirements across these applications and highlight the ones addressed by this dissertation.

High Scalability: The system should scale to large number of data streams and user queries. This necessarily means that the system should be designed to reduce infrastructure overload. In addition, the system should provide the required quality of service (throughput, response time) even in the presence of high load and high data volume.

Heterogeneity and Extensibility: With advances in sensing technologies, it has become possible to deploy different types of sensors on a large scale to derive actionable knowledge. There is also a need to use different types of sensors because a single sensing modality is often not sufficient to provide accurate situational knowledge.

Programming Abstraction: All the above example applications are inherently distributed. Building these applications is hard because of the challenges in distributed programming such as synchronization, buffer management, multi-threading, and performance. Moreover, the applications span multiple computer science disciplines and it is necessary to have common tools that would make building such applications easier.

Resource Management: Streaming vision processing applications are computationally demanding and need access to high performance computing resources. Resources need to be allocated and re-assigned dynamically based on observed output and application quality of service. Moreover, the resources may span multiple administrative domains and have different access policies. Harnessing these HPC resources efficiently is a key requirement of these applications.

Computation priority: As demonstrated in the distributed vehicle tracking application, different computations may have different priorities. The resource management system needs to consider these priorities in making run-time scheduling decisions.

One common characteristic of streaming applications is data fusion, whereby, data from multiple input sources is fused together to derive higher level information. For example, in a video-based surveillance application, data from multiple sources are fused in an application specific manner to identify interesting events. In this dissertation, I present a novel programming abstraction, called DFuse, for building these data fusion applications. The programming abstraction is extensible and addresses heterogeneity (by being programming language independent and supporting different kinds of data producers as well as data consumers). I also present a scalable architecture and algorithms for HPC resource management. I demonstrate that our resource management system automatically adapts under dynamic resource availability and application requirement changes. Considering computation priorities in

resource management is part of future work and not addressed in this dissertation.

Throughout the remaining chapters, I use distributed surveillance as the canonical example application in order to keep the discussion focused.

CHAPTER III

DFUSE PROGRAMMING ABSTRACTION

Developing streaming applications is challenging due to real-time application requirements as well as heterogeneity in resources spanning sensor network and HPC Clusters. Therefore, the first requirement is a uniform programming abstraction that hides the complexity of distributed programming and the underlying hardware while providing the flexibility to instantiate complex streaming application dataflow graphs.

In this chapter, I describe the DFuse programming abstraction that is designed to ease fusion application development. One common requirement for streaming applications is data fusion, whereby, data from multiple input sources is *fused* together to derive higher level information. The input source to a fusion function may be derived from the output of another fusion function; thereby forming a fusion function hierarchy.

Developing fusion applications is challenging in general because of the time-sensitive nature of the fusion operation, and the need for synchronization of the data from multiple streams. Since the applications are inherently distributed, they are typically implemented via distributed threads that perform fusion in a hierarchical manner. Thus, the application programmer has to deal with thread management, data synchronization, buffer handling, and exceptions (such as time-outs while waiting for input data for a fusion function) - all in a distributed fashion.

We have developed DFuse[52, 69], an architecture for programming fusion applications. Using the DFuse framework, application programmers need only implement the fusion functions and provide the dataflow graph (the relationships of fusion functions

to one another, as shown in Figure 5). The fusion API in the DFuse architecture subsumes issues such as data synchronization and buffer management that are inherent in distributed programming.

DFuse [52, 69] has been implemented and evaluated in the context of wireless ad hoc sensor network, but the design of the fusion APIs is general and applicable for any platform.

My main contributions in DFuse work are the design, implementation, and evaluation of the programming abstractions provided by DFuse. They are summarized below.

1. **Fusion API**: I design and implement a rich API that affords programming ease for developing complex data fusion applications. The API allows any synthesis operation on stream data to be specified as a fusion function, ranging from simple aggregation (such as min, max, sum, or concatenation) to more complex perception tasks (such as analyzing a sequence of video images). This is in contrast to current in-network aggregation approaches [57, 45, 42] that allow only limited types of aggregation operations as fusion functions.

2. **Fusion application development made easy**: DFuse allows the application to be represented simply as a set of fusion functions and their relationships to one another. Application writers do not have to worry about issues like data synchronization among fusion functions or mapping the fusion functions on to the network.

3. **Quantitative evaluation of the DFuse framework**: The evaluation includes micro-benchmarks of the primitives provided by the fusion API as well as measurement of the data transport in a tracker application.

In addition, DFuse has been used to evaluate various role assignment heuristics for placement of fusion functions in wireless ad hoc sensor network (WASN). The

main contributions (made by my colleagues using the DFuse framework with my collaboration) from that evaluation are as follows.

1. Distributed algorithm for deploying fusion points: An optimal placement is a mapping from the application fusion graph to the network nodes such that in-network traffic is minimized. We have developed a heuristic-based distributed algorithm for finding a *good* role assignment. The algorithm also accounts for dynamic changes in the WASN such as changes in the relative power levels of the nodes as well as node failures.

2. Evaluation of role assignment algorithms: Using an implementation of the fusion API on a wireless iPAQ farm coupled with an event-driven engine that simulates the WASN, we quantify the ability of the distributed algorithm to increase the longevity of the network with a given power budget of the nodes. For example, we show that the proposed role assignment algorithm increases the network lifetime by 110% compared to static placement of the fusion functions.

The rest of this chapter is organized as follows. Section 3.1 highlights the application context and the requirements. Section 3.2 analyzes fusion application requirements and presents the DFuse architecture. In Section 3.3, I describe how DFuse supports distributed data fusion. This is followed by the implementation details of the framework in Section 3.5 and its evaluation in Section 3.6. I also describe and summarize the key results from the role assignment heuristics in Section 3.4 for completeness. I then compare the framework with other existing and ongoing efforts in Section 3.8 and conclude with a summary of DFuse framework contributions in Section 3.9.

3.1 Application Context and Requirements

A fusion application has the following characteristics: (*i*) it is continuous in nature, (*ii*) it requires efficient transport of data from/to distributed sources/sinks, and (*iii*)

it requires efficient in-network processing of application-specified fusion functions. A data source may be a sensor (e.g., camera) or a stand-alone program; a data sink represents an end consumer and includes a human in the loop, an actuator (e.g., fire alarm), an application (e.g., data logger), or an output device such as a display; a fusion function transform the data streams (including aggregation of separate streams into a composite one) en route to the sinks. Thus, a fusion application is a directed task graph: the vertices are the fusion functions, and the edges represent the data flow (i.e., producer-consumer relationships) among the fusion points (cycles - if any - represent feedback in the task graph).

This formulation of the fusion application has a nice generality. It may be an application in its own right (e.g., video based surveillance). It allows hierarchically composing a bigger application (e.g., emergency response) wherein each component may itself be a fusion application (e.g., image processing of videos from traffic cameras). It allows query processing by overlaying a specific query (e.g., "show a composite video of all the traffic at the spaghetti junction") on to the task graph.

Consider for example, a video-based surveillance application. Cameras are deployed in a distributed fashion; the images from the cameras are filtered in some application-specific manner, and fused together in a form that makes it easy for an end user (human or some program) to monitor the area. The compute intensive part may analyze multiple camera feeds from a region to extract higher level information such as "motion", "presence or absence of a human face", or "presence or absence of any kind of suspicious activity". Figure 5 shows the task graph for this example application, in which filter and collage are fusion functions that transform input streams into output streams in an application specified manner. The fusion functions may result in contraction or expansion of data flows in the network. For example, the filter function selects images with some interesting properties (e.g., rapidly changing scene), and sends the compressed image data to the collage function. Thus, the filter

function is an example of a fusion point that does data contraction. The collage function uncompresses the images coming from possibly different locations. It combines these images and sends the composite image to the root (sink) for further processing. Thus, the collage function represents a fusion point that may do data expansion.

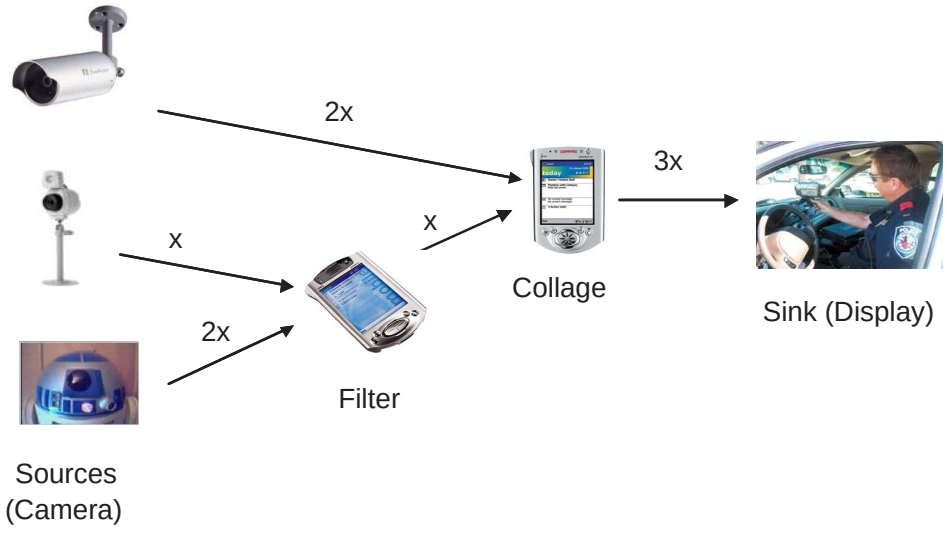


Figure 5: An example tracking application that uses distributed data fusion. Here, *filter* and *collage* are the two fusion points taking inputs from the cameras, and the face recognition algorithm is running at the end device. The numbers on edges of the figure show relative (expected) transmission rates of data sources and the fusion functions.

3.1.1 Architectural Assumptions

We have designed the DFuse architecture to cater to the evolving application needs and emerging technology trends. I made some basic assumptions about the execution environment in the design of DFuse APIs:

- The application level input to the architecture are:

    1. an application task graph consisting of the data flows and relationship among the fusion functions
    2. the code for the fusion functions (currently supported as C program binaries)

- For completeness, I list here all the assumptions that were made for role assignment heuristics.

    1. The application also provides a cost function that formalizes some application quality metric for the sensor network (e.g., keep the average node energy in the network the same).
    2. The task graph has to be mapped over a large geographical area. In the ensuing overlay of the task graph on to the real network, some nodes may serve as relays while others may perform the application-specified fusion operations.
    3. The fusion functions may be placed anywhere in the sensor network as long as the cost function is satisfied.
    4. All source nodes are reachable from the sink nodes.
    5. Every node has a routing layer that allows each node to determine the route to any other node in the network. This is in sharp contrast to most current day sensor networks that support all-to-sink style routing. However, the size of the routing table in every node is only proportional to the size of the application task graph (to facilitate any network node in the ensuing overlay to communicate with other nodes hosting fusion functions) and not the physical size of the network.
    6. The routing layer exposes information (such as hop-count to a given destination) that is needed for making mapping decisions in the DFuse architecture.

3.2 DFuse Architecture Components

Figure 6(A) shows the components of the DFuse architecture. There are two components to this architecture: fusion module, and placement module. Fusion module is

my main contribution in this dissertation.

From an application perspective, there are two main concerns:

1. How do we develop the fusion application? The fusion code itself (e.g., "motion detection" over a number of incoming video streams and producing a digest) is in the purview of the application developer. However, there are a number of systems issues that have to be dealt with before a fusion operation can be carried out at a given node including: (*i*) providing the "plumbing" from the sources to the fusion point; (*ii*) ensuring that all the inputs are available; (*iii*) managing the node resources (CPU and memory) to enhance performance, and (*iv*) error and failure handling when some sources are non-responsive. I have designed a fusion module with a rich API that deals with all of the above issues. I describe this module in Section 3.3.

2. How do we generate an overlay of the task graph on to the sensor network? As we mentioned earlier, some nodes in the overlay will act as relays and some will act as fusion points. Since the application is dynamic (sources/sinks may join/leave as dictated by the application, new tasks may be created, etc.), and the physical network is dynamic (sources/sink may fail, intermediate nodes may run out of energy, etc.) this mapping is not a one-time job. After an initial mapping re-evaluation of the mapping (triggered by changes in the application or the physical infrastructure) will lead to new assignment and re-assignment of the nodes and the roles they play (relay vs. fusion). We have designed a placement module that embodies a role assignment algorithm that deals with the above issues. This work was primarily done by others and I summarize it in Section 3.4 for completeness.

3.3 Fusion Module

DFuse provides a package of high-level abstractions for supporting fusion operations in stream-oriented environments. This package, called **Fusion Channels**, is conceptually language and platform independent.

The Fusion Channels package aims to simplify the application of programmer-supplied transformations to correlated sets of input items from sequenced input streams, producing a (possibly shared) output stream of "fused items". It does this by providing a high-level API for creating, modifying, and manipulating fusion points that subsumes certain recurring concerns (failure, latency, buffer management, prefetching, mobility, sharing, concurrency, etc.) common to fusion environments. Only a subset of the capabilities in the Fusion Channels package are currently used by DFuse. Figure 6(B) shows the internal structure of the fusion module. The capabilities of fusion API are described in the following subsections.

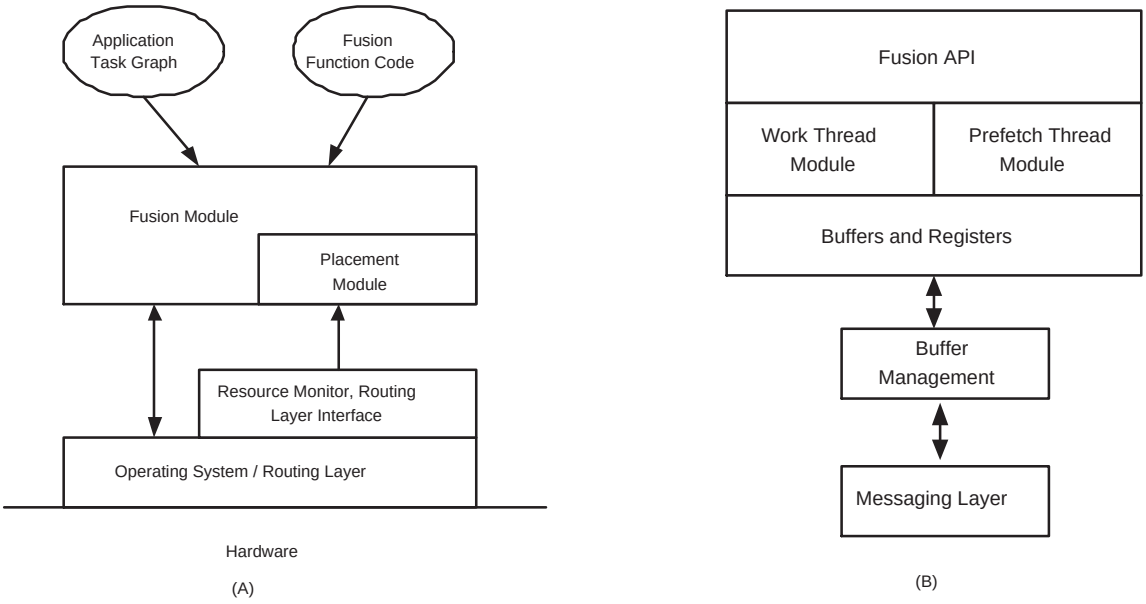


Figure 6: (A) DFuse architecture - a high-level view. (B) Fusion module components.

3.3.1 Structure management

This category of capabilities primarily handles "plumbing" issues. The fundamental abstraction in DFuse that encapsulates the fusion function is called a *fusion channel*. A fusion channel is a named, global entity that abstracts a set of inputs and encapsulates a programmer-supplied fusion function. Inputs to a fusion channel may come from the node that hosts the channel or from a remote node. Item fusion is automatic and is performed according to a programmer-specified policy either on request (demand-driven, lazy, pull model) or when input data is available (data-driven, eager, push model). Items are fused and accessed by timestamp (usually the capture time of the incoming data items). An application can request an item with a particular timestamp or by supplying some wildcard specifiers supported by the API (such as earliest item, latest item). Requests can be blocking or non-blocking. To accommodate failure and late arriving data, requests can include a minimum number of inputs required and a timeout interval. Fusion channels have a fixed capacity specified at creation time. Finally, inputs to a fusion channel can themselves be fusion channels, creating fusion networks or pipelines. Using a standard or programmer-supplied protocol, a fusion channel may be migrated on demand to another node of the network. This feature is essential for supporting the role assignment functionality of the placement module. Upon request from an application, the state of the fusion channel is packaged and moved to the desired destination node by the fusion module. The fusion module handles request forwarding for channels that have been migrated.

- Structure Management APIs

```
channel = createFC(inputs, fusion_function)
result = destroyFC(channel)
channel_connection = attachFC(channel)
result = detachFC(channel_connection)
```

```
item = get/putFCItem(channel_connection, attributes)
result = consumeFCItem(channel, attributes)
inputs = get/setFCInputs(channel, new_inputs)
inputs = addFCInput(channel, input)
inputs = removeFCInput(channel, input_index)
location = getFCLocation(channel)
result = moveFC(channel, new_location)
result = moveFC(channel, new_location, protocol)
```

CreateFC, destroyFC, attachFC, detachFC APIs allow for fusion channel creation, destruction, attachment, and detachment respectively. An application typically creates a fusion channel so that any node in the dataflow graph can subsequently attach to it for performing I/O operations. The get/PutFCItem APIs allow the application to perform I/O operations on the channel. The consumeFCItem APIs allows the application to specify certain items as consumed so that they can be removed from the system by a garbage collection algorithm. The next set of APIs (addFCInput, removeFCInput) supports dynamic configuration of a fusion function by adding or removing inputs to the channel. The final set of structure management APIs (getFCLocation, MoveFC) supports fusion point migration from one location to another driven by the application.

3.3.2 Correlation control

This category of capabilities primarily handles specification and collection of "correlation sets" (related input items supplied to the fusion function). Fusion requires identification of a set of correlated input items. A simple scheme is to collect input items with identical application-specified sequence numbers or virtual timestamps (which may or may not map to real-time depending on the application). Fusion functions may declare whether they accept a variable number of inputs and, if so, indicate

bounds on the correlation set size. Correlation may involve collecting several items from each input (for example, a time-series of data items from a given input). Correlation may specify a given number of inputs or correlate all arriving items within a given time interval. Most generally, correlation can be characterized by two programmer-supplied predicates. The first determines if an arriving item should be added to the correlation set. The second determines if the collection phase should terminate, passing the current correlation set to the programmer-supplied fusion function.

- Correlation Control APIs

```
params = get/setFCCorrelation(channel, correlation_params)
params include:
    min, max correlation set size
    correlate by timestamp?
    correlation ranges for temporal correlation (per input)
    discard late items?
    correlation predicates:
        boolean addItemToCorrelationSet(channel, item, set)
        boolean activateFusionFunction(channel, set)
```

The get/setFCCorrelation APIs control how items are correlated to perform fusion operation. The min, max correlation set size parameter control partial fusion. For example, if the minimum number of inputs are available within a timeout period, the fusion module performs partial fusion on those inputs instead of waiting for all the inputs to arrive. Similarly, the max parameter controls the maximum number of inputs that can be fused together in a single fusion operation. Items could be fused based on timestamp or based on temporal correlation per input channel. Temporal correlation per input channel supports aggregations of data across a single input channel whereas correlation across channels is used to perform data fusion from multiple data sources.

The correlation parameters also control whether late arriving items can be discarded or added to subsequent fusion operation. Finally, addItemToCorrelationSet is used to add new items to correlation set and activateFusionFunction is used to activate the fusion function when sufficient number of input data items are available.

3.3.3 Memory Management

This category of capabilities primarily handles caching, prefetching, and buffer management. Typically, inputs are collected and fused (on-demand) when a fused item is requested. For scalable performance, input items are collected (requested) in parallel. Requests on fusion pipelines or trees initiate a series of recursive requests. To enhance performance, programmers may request items to be prefetched and cached in a prefetch buffer once inputs are available. An aggressive policy prefetches (requests) inputs on-demand from input fusion channels. Buffer management deals with sharing generated items with multiple potential consumers and determining when to reclaim cached items' space.

- Caching, Prefetching, Buffer management APIs

```
size = get/setFCPrefusionBufferSize(channel)
policy = get/setFCFusionPolicy(channel, fusion_policy)
policy = get/setFCPrefusionBufferExpiry(channel, expiry_policy)
```

The get/setFCPrefusionBufferSize is used to control the buffer size of fusion channel. The get/setFCFusionPolicy control the fusion policy (on-demand or prefetching). The buffer expiry policy is controlled by the get/setFCFusionPolicy APIs. Items in the buffer can be expired on a timeout or based on more recent data arrival.

3.3.4 Failure/latency handling

This category of capabilities primarily allows the fusion points to perform partial fusion, i.e., fusion over an incomplete input correlation set. If data from all input

channels are not available within a specified timeout period, the application may still be able to function correctly by using partial fusion. For example, in a distributed surveillance application, multiple camera sources monitor the same environment. Therefore, it is not essential to get the data from all the input sources for making higher level inferences. Therefore, partial fusion deals with node failure and communication latency that are common, and often indistinguishable, in distributed networks. Fusion functions capable of accepting a variable number of input items may specify a timeout on the interval for correlation set collection. Late arriving items may be automatically discarded or included in subsequent correlation sets.

If the correlation set contains fewer items than needed by the fusion function, an error event occurs and a programmer-supplied error handler is activated. Error handlers and fusion functions may produce special *error items* as output to notify downstream consumers of errors. Fused items include meta-data indicating the inputs used to generate an item in the case of partial fusion. Applications may use the structure management API functions to remove the faulty input if necessary.

- Failure and Latency Handling APIs

```
timeout = get/setFCCorrelationTimeout(channel, timeout)
policy = get/setFCCorrelationTimeoutPolicy(channel, timeout_policy)
item = get/putFCErrorItem(channel, error_item)
```

These APIs are used to control the timeout, timeout policy, and error handling. The timeout policy could specify that partial fusion can be used or it may specify dropping of available data items in case of timeout. It may also specify waiting for all the input items to be available before performing fusion. The error items are used to invoke programmer supplied error handler and notify consumers of the fusion channel.

3.3.5 Status and feedback handling

This category of capabilities primarily allows interaction between fusion functions and data sources such as sensors that supply status information and support a command set (for example, activating a sensor or altering its mode of operation - such devices are often a combination of a sensor and an actuator). We have observed that application-sensor interactions tend to mirror application-device interactions in operating systems. Sources such as sensors and intermediate fusion points report their status via a "status register".[1] Intermediate fusion points aggregate and report the status of their inputs along with the status of the fusion point itself via their respective status registers. Fusion points may poll this register or access its status. Similarly, sensors that support a command set (to alter sensor parameters or explicitly activate and deactivate) should be controllable via a "command" register. The specific command set is, of course, device specific but the general device driver analogy seems well-suited to control of sensor networks.

- Status and Feedback Handling APIs

```
status = get/putFCStatus(channel, status, include_inputs?)
command = get/putFCCommand(channel, command, propagate?)
```

These APIs are used by the fusion points to control the data sources and provide their status. Intermediate fusion points may be asked to include the status of their inputs as well thereby generating downstream calls to the API on their inputs. Similarly, a command (such as change camera resolution) can be propagated from an intermediate fusion point to all the data sources.

[1] A register is a communication abstraction with processor register semantics. Updates overwrite existing values, and reads always return the current status.

3.3.6 Computation management

This category of capabilities primarily handles the specification, application, and migration of fusion functions. The fusion function is a programmer-supplied code block that takes as input a set of timestamp-correlated items and produces a fused item (with the same timestamp) as output. A fusion function is associated with the channel when created. It is possible to dynamically change the fusion function after channel creation, to modify the set of inputs, and to migrate the fusion point. Using a standard or programmer-supplied protocol, a fusion channel may be migrated on demand to another node of the network. This feature is essential for supporting the role assignment functionality of the placement module. Upon request from an application, the state of the fusion channel is packaged and moved to the desired destination node by the fusion module. The fusion module handles request forwarding for channels that have been migrated.

- Computation Management APIs

```
function = get/setFCFunction(channel, fusion_function)
handler = get/setFCEventHandler(channel, event, handler)
source = get/setFCAsynchInput(channel, source)
```

The above APIs are used to control the fusion functions associated with a channel and the action taken on various events. The setFCFunction API is used to change the fusion function associated with a channel at runtime or migrate a fusion point. Similarly, get/setFCEventHandler, and get/setFCAsyncInput APIs are used to control the input to the fusion channel and control the way fusion channel handle application specific events.

3.4 Placement Module

Other co-authors in DFuse contributed to the design, implementation, and evaluation of placement module and various role assignment heuristics. I summarize it here for completeness.

The placement module is responsible for creating an overlay of the application task graph on to the physical network that best satisfies an application-specified cost function. A network node can play one of the three roles: end point (source or sink), relay, or fusion point. In our model, the end points are determined by the application. The placement module embodies a distributed role assignment algorithm that manages the overlay network, dynamically assigning fusion points to the available nodes in the network.

3.4.1 Placement Requirements in Wireless Ad-Hoc Sensor Network

The role assignment algorithm has to be aware of the following aspects of a Wireless Ad-hoc Sensor Network (WASN).

Node Heterogeneity: A given node may take on multiple roles. Some nodes may be resource rich compared to others. For example, a particular node may be connected to a permanent power supply. Clearly, such nodes should be given more priority for taking on transmission-intensive roles compared to others.

Power Constraint: A role assignment algorithm should minimize data communication since data transmission and reception expend more power than computation activities in wireless sensor networks [43]. Intuitively, since the overall communication cost is impacted by the location of data aggregators, the role assignment algorithm should seek to find a suitable placement for the fusion points that minimizes data communication.

Dynamic Behavior: There are two sources of dynamism in a WASN. First, the application may exhibit dynamism due to the physical movement of end points or change in the transmission profile. Second, there could be node failures due to environmental conditions or battery drain. So far as the placement module is concerned, these two conditions are equivalent. In either case, the algorithm needs to find a new mapping of the task graph onto the available network nodes.

Our heuristic is based on a simple idea: first perform a naive assignment of roles to the network nodes, and then allow every node to decide locally if it wants to transfer the role to any of its neighbors. Upon completion of the naive assignment phase, a second phase of role transfer begins. A node hosting any fusion point role, checks if one of its neighbor nodes can host that role better using a cost function to determine the "goodness" of hosting a particular role. If a better node is found then a role transfer is initiated. Since all decisions are taken locally, every node needs to know only as much information as is required for determining the goodness of hosting a given role for a given application task graph. For example, if the cost function is based upon the remaining power level at the host, every node needs to know only its own power level. For further details on the role assignment algorithms, please refer to DFuse [52, 69].

3.5 Implementation

DFuse is implemented as a multi-threaded runtime system, assuming infrastructure support for timestamping data produced from different sensors, and a reliable transport layer for moving data through the network. Multi-threading the runtime system enhances opportunities for parallelism in data collection and fusion function execution for streaming tasks. The infrastructural assumptions can be satisfied in various ways. As we mentioned earlier, the timestamps associated with the data can be virtual or real. Virtual timestamping has several advantages, the most important of which is

the fact that the timestamp can serve as a vehicle for propagating the causality between raw and processed data from a given sensor. Besides, virtual timestamps allows an application to choose the granularity of real-time interval for chunking streaming data. Further, the runtime overhead is minimized since there is no requirement for global clock synchronization, making virtual time synchrony attractive.

Assuming above infrastructure support, implementing DFuse consists of implementing a multi-threaded architecture for the fusion module that supports the basic fusion API calls (Section 3.3), and the other associated optimizations such as prefetching.

The infrastructural requirements are met by a programming system called Stampede [65, 7]. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through channels. Stampede also provides registers with full/empty synchronization semantics for inter-thread signaling and event notification. The threads, channels, and registers can be launched anywhere in the distributed system, and the runtime system takes care of automatically garbage collecting the space associated with obsolete items from the channels.

3.5.1 Data Fusion Module

I have implemented the fusion architecture in C as a layer on top of the Stampede runtime system. All the buffers (input buffers, fusion buffer, and prefetch buffer) are implemented as Stampede channels. Since Stampede channels hold timestamped items, it is a straightforward mapping of the fusion attribute to the timestamp associated with a channel item. The Status and Command registers of the fusion architecture are implemented using the Stampede register abstraction. In addition to these Stampede channels and registers that have a direct relationship to the elements of the fusion architecture, the implementation uses additional Stampede channels and threads. For instance, there are prefetch threads that gather items from the input

buffers, fuse them, and place them in the prefetch buffer for potential future requests. This feature allows latency hiding but comes at the cost of potentially wasted network bandwidth and hence energy (if the fused item is never used). Although this feature can be turned off, we leave it on in our evaluation and ensure that no such wasteful communication occurs. Similarly, there is a Stampede channel that stores requests that are currently being processed by the fusion architecture to eliminate duplication of work.

The *createFC* call from an application thread results in the creation of all the above Stampede abstractions in the address space where the creating thread resides. An application can create any number of fusion channels (modulo system limits) in any of the nodes of the distributed system. An *attachFC* call from an application thread results in the application thread being connected to the specified fusion channel for getting fused data items. For efficient implementation of the *getFCItem* call, a pool of worker threads is created in each node of the distributed system at application startup. These worker threads are used to satisfy getFCItem requests for fusion channels created at this node. Since data may have to be fetched from a number of input buffers to satisfy the getFCItem request, one worker thread is assigned to each input buffer to increase the parallelism for fetching the data items. Once fetching is complete, the worker thread rejoins the pool of free threads. The worker thread to fetch the last of the requisite input items invokes the fusion function and puts the resulting fused item in the fusion buffer. This implementation is performance-conscious in two ways: first, there is no duplication of fusion work for the same fused item from multiple requesters; second, fusion work itself is parallelized at each node through the worker threads.

The duration to wait on an input buffer for a data item to be available is specified via a policy flag to the *getFCItem*. For example, if *try_for_time_delta* policy is specified, then the worker thread will wait for time *delta* on the input buffer. On the other

hand, if *block* policy is specified, the worker thread will wait on the input buffer until the data item is available. The implementation also supports *partial fusion* in case some of the worker threads return with an error code during fetch of an item. Taking care of failures through partial fusion is a very crucial component of the module since failures and delays can be common in wide area distributed systems.

As we mentioned earlier, Stampede does automatic reclamation of storage space of data items in channels. Stampede garbage collection uses a global lower bound for timestamp values of interest to any of the application threads (which is derived from a per-thread state variable called thread virtual time). Our fusion architecture implementation leverages this feature for cleaning up the storage space in its internal data structures (which are built using Stampede abstractions).

3.5.2 Placement Module

I briefly describe here the implementation of placement module for completeness. Stampedes runtime system sits on top of a reliable UDP layer in Linux. Therefore, there is no support for adaptive multi-hop ad hoc routing in the current implementation. Further, there is no support for gathering real health information of the nodes. For the purposes of evaluation, we have adopted a novel combined implementation/simulation approach. The fusion module is a real implementation on a farm of iPAQs as detailed in the previous subsection. The placement module is an event-driven simulation of the role assignment algorithm. It takes an application task graph and the network topology information as inputs, and generates an overlay network, wherein each node in the overlay is assigned a unique role of performing a fusion operation. It models the health of the sensor network nodes. It currently assumes an ideal routing layer (every node knows a route to every other node) and an ideal MAC layer (no contention). However, it should be clear that these assumptions are not inherent in the DFuse architecture; they are made only to simplify the simulator

implementation and evaluation.

We have implemented an interface between the the fusion module implementation and the placement module simulation. This interface facilitates (*i*) collecting the actual data rates of the sensor nodes experienced by the application running on the implementation and reporting them to the placement module simulation, and (*ii*) communicating to the fusion module (and effecting through the DFuse APIs) dynamic task graph instantiation, role changes based on the health of the nodes, and fusion channel migration.

3.6 Fusion API Performance Evaluation

We have performed an evaluation of the fusion and placement modules of the DFuse architecture at two different levels: micro-benchmarks to quantify the overhead of the primitive operations of the fusion API including channel creation, attachments/detachments, migration, and I/O; ability of the placement module to optimize the network given a cost function.

My primary contribution is in demonstrating that Fusion API has reasonable overhead and it performs better than a naive implementation that doesn't provide the functionalities such as parallel data fetching. Also through scalability study, I show that the API overhead increases independent of data size as number of channels is increased (primarily because of parallel data fetching).

The DFuse framework is also evaluated in a Sensor Network environment. This work was primarily done by my colleagues. I summarize briefly the experimental results conducted on a set of IPAQs to demonstrate the benefits of Fusion APIs and role assignment algorithms in sensor network.

3.6.1 Micro Measurements

In this section, I report preliminary performance results of the fusion architecture. The experimental platform for this work is a cluster of SMP nodes running Linux.

The hardware consists of 17 Dell 8450 servers each with eight 550MHz Pentium III Xeon CPUs, 2MB of L2 cache per CPU and 4GB of memory per node. The 8450 uses the Intel ProFusion chipset which provides two 64-bit/100MHz system (front-side) busses, one for each bank of four CPUs. The nodes are interconnected with gigabit Ethernet through a dedicated switch. The operating system is Linux with the 2.4.9 kernel. The system scheduler in this kernel is oblivious to the 8450s split system bus. The compiler is GCC version 2.96 with optimization set to -O2.

I perform a set of micro measurements to quantify the overhead of the getFCItem API (which is built on top of Stampede) by comparing it to an implementation that performs the fusion directly using Stampede channels (i.e., without the fusion library).

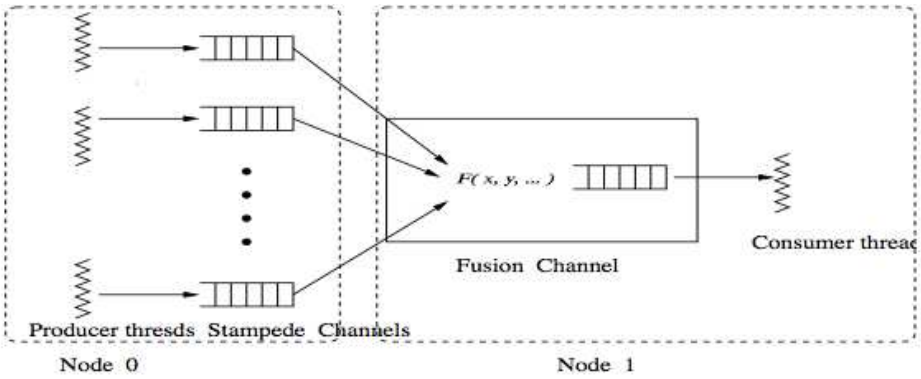


Figure 7: Measuring the latency of getFCItem API

Figure 7 shows the setup for a series of micro experiments. A simple fusion application that spans two nodes is shown. A fusion channel is collocated with a consumer on Node 1. The inputs that need to be fused are produced on Node 0 by producer threads. The consumer calls fusion get to fuse the data items produced on Node 0. The latency for this operation is measured in this experiment.

Figure 8 describes a comparable experimental setup designed using Stampede channels only without the fusion library. This setup performs the exact same fusion application as in Figure 7. The main difference between the two experimental setups is that the fusion library implicitly uses thread parallelism for fetching the inputs for

fusion in Figure 7, whereas the straight Stampede implementation in Figure 8 naively uses a single thread that has to sequentially fetch all the inputs before fusing them. We conduct three micro measurements.

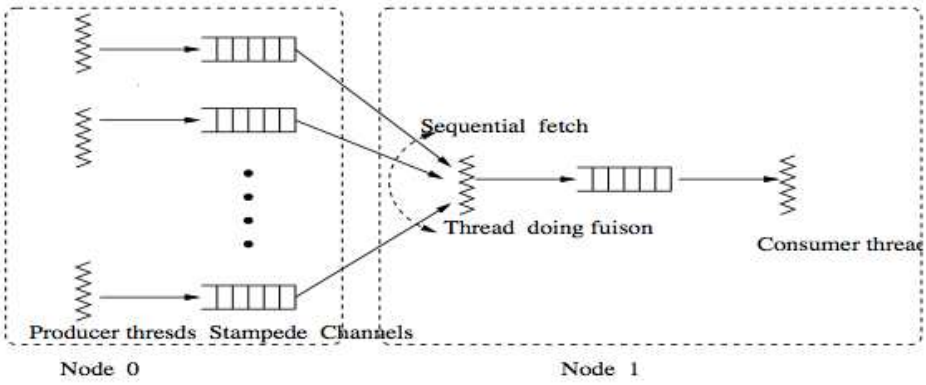


Figure 8: Experiment setup without fusion library

Experiment 1: In this experiment we wish to measure the overhead of the fusion architecture. We use only a single input. This helps us quantify the exact overhead of the fusion architecture that has been implemented on top of Stampede. We study the latency of the getFCItem call as a function of increasing data size. This is compared against the scenario in Figure 8 that is implemented without the fusion library. Figure 9 plots the latency in millisecond, against the data size (given in bytes). The latency when using the fusion library is little more than without it. The difference is also plotted in the same graph. For a data size of 1.28MB, the fusion library adds an overhead of 1.7 millisecond which represents a 21% increase in latency over vanilla Stampede.

Experiment 2: In this experiment we wish to measure the performance advantage that the fusion library gives when there are multiple inputs that need to be fused. The main source of expected performance improvement is the thread parallelism used in the fusion library for fetching the inputs in parallel for fusion compared to the naive implementation. In this experiment we keep the data size constant (1 KB) and vary the number of input channels (plotted on the X axis).

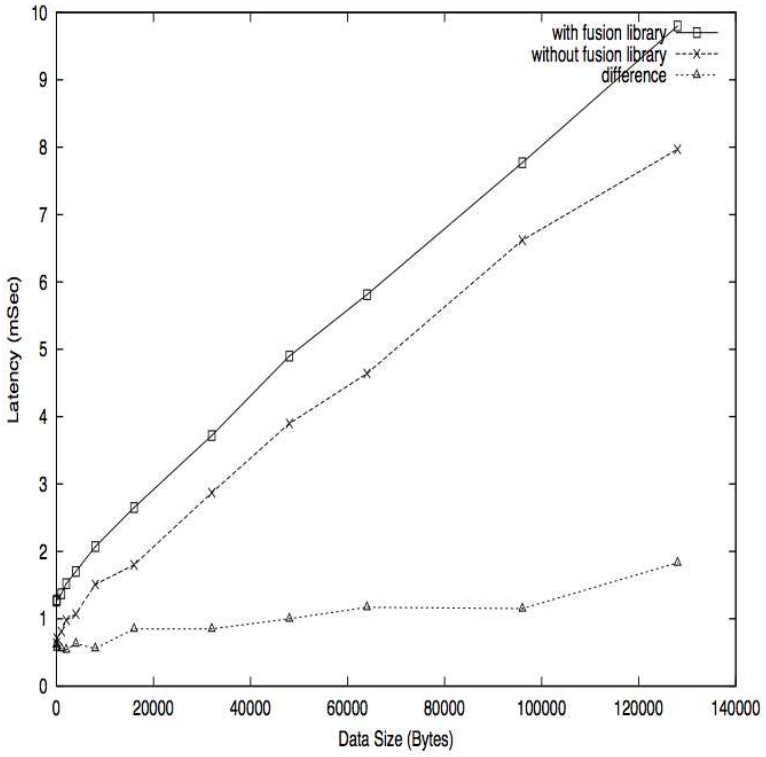


Figure 9: Overhead of getFCItem API

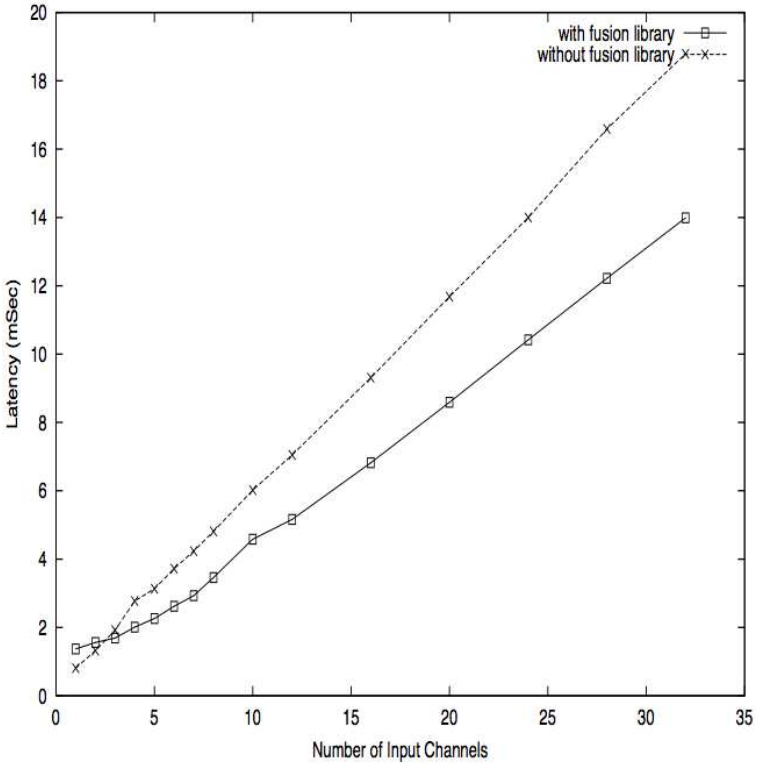


Figure 10: Effect of Parallel Fetch with Collocated Producers

The latencies (on Y axis) for the two experimental setups are shown in Figure 10. As can be seen, as the number of inputs grows, the fusion library version starts performing better. The size of the thread pool was kept at 8 (Since all our experiments are carried out on 8-way SMPs, 8 threads give maximum parallelism). For 32 input channels, the fusion API has a latency of 14 milliseconds, while the latency without the fusion library is almost 19 milliseconds. Therefore, the fusion library performs about 26% better in this experiment.

Experiment 3: This experiment is a slight variation of Experiment 2. The setup is the same except that each producer thread runs on a different node. Figure 11 illustrates the performance figures for this experiment. The X-axis represents number of input channels and the Y-axis represents latency as before. Data size is kept constant at 1KB. In this experiment, we varied number of input channels up to 10 (and not 32, as in Experiment 2) as we were limited by the number of machines. For 10 input channels, the fusion library latency is 4.5 milliseconds, while the naive implementation latency is 6.5 milliseconds. This experiment illustrates that our system gives about 30% better performance when all the producers are on different nodes.

3.6.2 Scalability with Number of Inputs

We estimate the expected improvement in scalability as follows. The total latency of obtaining a fused item is the sum of time units required to fetch and apply the fusion function. Fusion function time includes the computational time of the function, local copy into the fusion buffer and then final hand off to the consumer thread. The fusion library improves over the naive implementation by reducing the time required to fetch items (through parallelism). This is illustrated by a comparative observation of the latency variation for data size 1KB and 2 KB. Figure 12 plots the latency for data size of 2KB. The slope of the line representing the fusion library remains approximately the same in both Figure 11 and Figure 12.

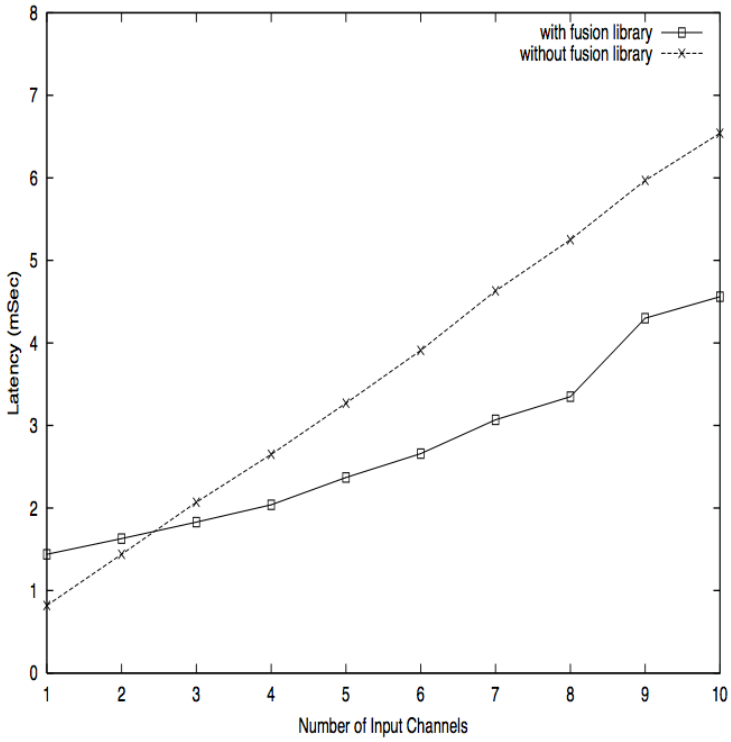


Figure 11: Effect of Parallel Fetch with Remote Producers (datasize=1KB)

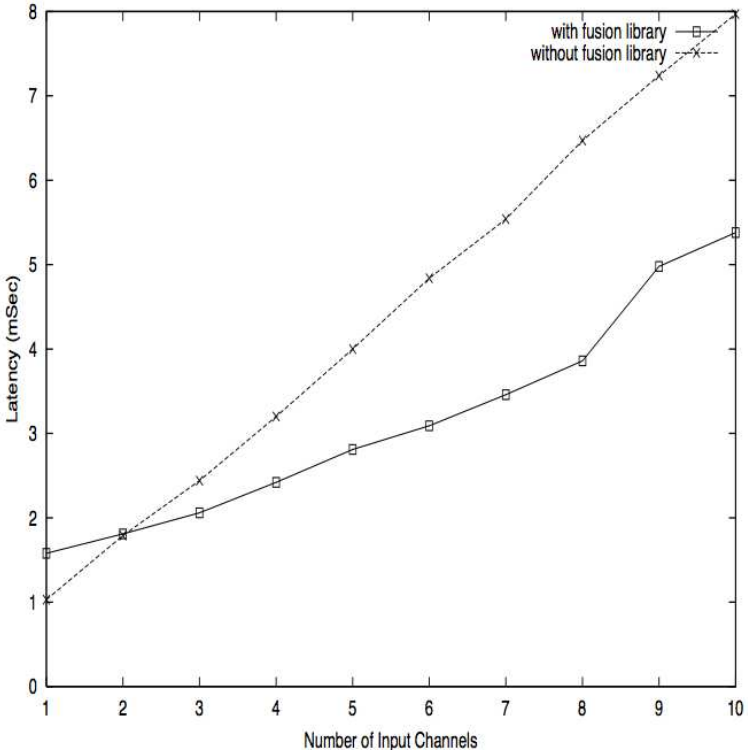


Figure 12: Effect of Parallel Fetch with Remote Producers (datasize=2KB)

To make this observation clear, in Figure 11, the latency (with fusion library) varies from 1.4 milliseconds to 3.4 milliseconds as the number of input channels varies from 1 to 8, thus giving an average slope of 0.29. In Figure 12, in a similar way the slope is approximately 0.31. Without the fusion library, the slope for Figure 11 is 0.63, while in Figure 12, i.e., data size 2KB, the slope is 0.78. This comparison shows that the fusion library scales better than a naive implementation.

In summary, the experiments demonstrate that getFCItem API provides about 30% better performance over a naive implementation when producers are distributed on different nodes. The performance benefit comes from parallel data fetching across multiple input channels. Also, I demonstrate that the API is scalable as the number of input data sources to a fusion channel increases.

In the next section, I demonstrate similar performance results obtained on an IPAQ. Also, I summarize the results from the evaluation of different role assignment algorithms. This part of the work was done by my colleagues with help from me and is summarized here for completeness[2].

3.7 Performance Results in Sensor Network

This experimental setup uses a set of wireless iPAQ 3870s running Linux "familiar" distribution version 0.6.1 together with a prototype implementation of the fusion module discussed in section 3.5.

Although the evaluations were conducted using limited capabilities of IPAQ, the experimental results demonstrate that the overhead of the APIs is acceptable in the context of wireless sensor network.

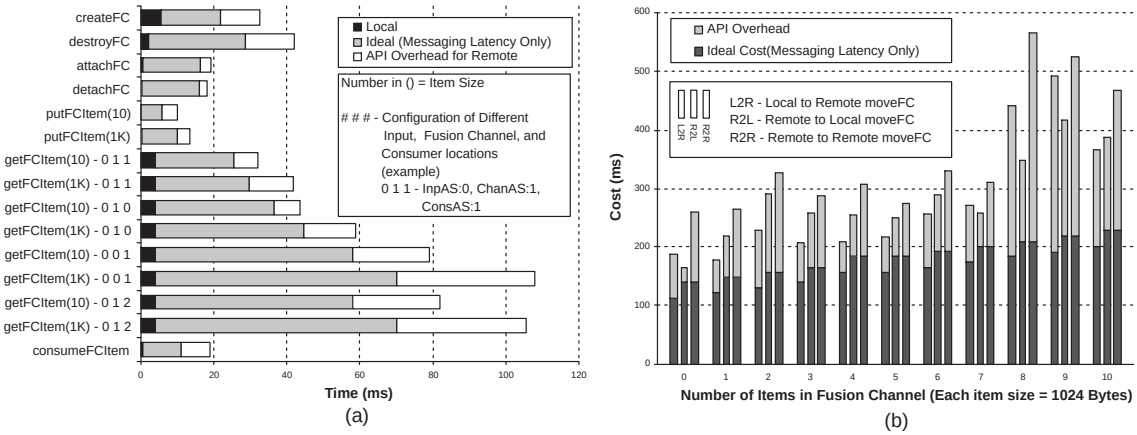


Figure 13: (a) Fusion Channel APIs' cost (b) Fusion channel migration (moveFC) cost

3.7.1 Fusion API Measurements

Figure 13 shows the cost of the DFuse API. In part (a), each API cost has 3 fields - local, ideal, and API overhead. Local cost indicates the latency of operation execution without any network transmission involved, ideal cost includes messaging latency only, and API overhead is the subtraction of local and ideal costs from actual cost on the iPAQ farm. Ideally, the remote call is the sum of messaging latency and local cost. Fusion channels can be located anywhere in the network. Depending on the location of the fusion channels input(s), fusion channel, and consumer(s), the minimum cost varies because it can involve network communications. *getFCItem* is the most complex case, having four different configurations and costs independent of the item sizes being retrieved. For part (a), we create fusion channels with capacity of ten items and one primitive Stampede channel as input. Reported latencies are the average of 1000 iterations.

On our iPAQ farm, netperf [64] indicates a minimum UDP roundtrip latency of 4.7ms, and from 2-2.5Mbps maximum unidirectional streaming TCP bandwidth. Table 3.7.1 depicts how many round trips are required and how many bytes of overhead

[2]Please note that most of the text in Section 3.7 comes from a paper co-authored by me with my colleagues [69].

exist for DFuse operations on remote nodes.

Table 1: Number of round trips and message overhead of DFuse. See Figure 13 for `getFCItem` and `moveFC` configuration legends.

API	Round Trips	Overhead (bytes)	API	Round Trips	Overhead (bytes)
`createFC`	3	596	`getFCItem`(1K) - 0 1 0	6	3112
`destroyFC`	5	760	`getFCItem`(10) - 0 0 1	10	1738
`attachFC`	3	444	`getFCItem`(1K) - 0 0 1	10	4780
`detachFC`	3	462	`getFCItem`(10) - 0 1 2	10	1738
`putFCItem`(10)	1	202	`getFCItem`(1K) - 0 1 2	10	4780
`putFCItem`(1K)	1	1216	`consumeFCItem`	2	328
`getFCItem`(10) - 0 1 1	4	662	`moveFC`(L2R)	20	4600
`getFCItem`(1K) - 0 1 1	4	1676	`moveFC`(R2L)	25	5360
`getFCItem`(10) - 0 1 0	6	1084	`moveFC`(R2R)	25	5360

From these measurements, we show messaging latency values in Figure 13(a) for ideal case costs on the farm. We calculate these ideal costs by adding latency per round trip and the cost of the transmission of total bytes, presuming 2Mbps throughput. Comparing these ideal costs in Figure 13(a) with the actual total cost illustrates reasonable overhead for our DFuse API implementation. The maximum cost of operations on a local node is 5.3ms. For operations on remote nodes, API overhead is less than 74.5% of the ideal cost. For operations with more than 20ms observed latency, API overhead is less than 53.8% of the ideal cost. This figure also illustrates that messaging constitutes the majority of observed latency of API operations on remote nodes. Note that ideal costs do not include additional computation and synchronization latencies incurred during message handling.

The placement module may cause a fusion point to migrate across nodes in the sensor fusion network. Migration latency depends upon many factors: the number of inputs and consumers attached to the fusion point, the relative locations of the node where `moveFC` is invoked to the current and resulting fusion channel, and amount of data to be moved. Our analysis in Figure 13(b) assumes a single primitive stampede channel input to the migrating fusion channel, with only a single consumer. Part (b) shares the same ideal cost calculation methodology as part (a). Our observations show that migration cost increases with number of input items and that migration from a remote to a remote node is more costly than local to remote or remote to local migration for a fixed number of items. Reported latencies are averages over 300 iterations for part (b).

The dynamic application requirement may require a fusion point to migrate across nodes in the fusion network. Migration latency depends upon many factors: the number of inputs and consumers attached to the fusion point, the relative locations of the node where *moveFC* is invoked to the current and resulting fusion channel, and amount of data to be moved. Our analysis in Figure 13(b) assumes a single primitive stampede channel input to the migrating fusion channel, with only a single consumer. Part (b) shares the same ideal cost calculation methodology as part (a). Our observations show that migration cost increases with number of input items and that migration from a remote to a remote node is more costly than local to remote or remote to local migration for a fixed number of items. Reported latencies are averages over 300 iterations for part (b).

3.7.2 Placement Algorithm Measurements

I briefly summarize evaluation of placement algorithms for completeness here.

To verify the design of the fusion module and placement algorithm, we have implemented the tracker application (Figure 5) using the fusion API and deployed it on

the iPAQ farm.

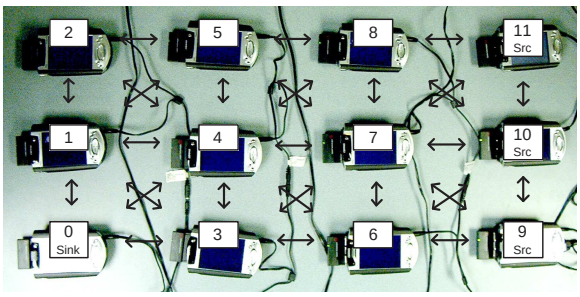


Figure 14: iPAQ Farm Experiment Setup. An arrow represents that two iPAQs are mutually reachable in one hop.

Figure 14 shows the topological view of the iPAQ farm used for the tracker application deployment. It consists of twelve iPAQ 3870s configured identically to those in the measurements above. Node 0, where node i is the iPAQ corresponding to ith node of the grid, acts as the sink node. Nodes 9, 10, and 11 are the iPAQs acting as the data sources. The location of *filter* and *collage* fusion points are guided by the placement module.

The placement module simulator runs on a separate desktop in synchrony with the fusion module. At regular intervals, it collects the transmission details (number of bytes exchanged between different nodes) from the farm. It uses a simple power model to account for the communication cost and to monitor the power level of different nodes. If the placement module decides to transfer a fusion point to another node, it invokes the `moveFC` API to effect the role transfer.

For transmission rates, we have tuned the tracker application to generate data at consistent rates as shown in Figure 5, with x equal to 6KBytes per minute. This is equivalent to a scenario where cameras scan the environment once every minute, and produce images ranging in size from 6 to 12KBytes after compression.

The network is organized as a grid shown in Figure 14. For any two nodes, the routing module returns the path with a minimum number of hops across powered nodes. To account for power usage at different nodes, the placement module uses a

simple approach. It models the power level at every node, adjusting them based upon the amount of data a node transmits or receives. The power consumption corresponds to ORiNOCO 802.11b PC card specification. Our current power model only includes network communication costs. After finding a naive tree, the placement algorithm runs in optimization phase for two seconds. The length of this period is tunable and it influences the quality of mapping at the end of the optimization phase. During this phase, fusion nodes wake up every 100ms to determine if role transfer is indicated by the cost function. After optimization, the algorithm runs in maintenance phase until the network becomes fragmented (some consumer cannot reach one of its inputs). During the maintenance phase, role transfer decisions are evaluated every 50 seconds. The role transfers are invoked only when the health improves by a threshold of 5%.

I am summarizing the results from the experiments here. The results show that the choice of cost function should be dependent upon application context and network condition. If, for an application, role transfer is complex and expensive, but network power variance is not an issue, then one particular cost function should be preferred. However, if network power variance needs to be minimized and role transfer is inexpensive, another cost function is preferable. Also we show that a distributed role assignment algorithm increases the network lifetime by 110% compared to static placement of the fusion functions in the experiments.

3.7.3 Discussion

By running the application and role assignment modules separately, we have simplified our evaluation approach. This approach has some disadvantages such as limited ability to communicate complex resource monitoring results. Transferring every detail of the running state from the fusion module to the placement module is prohibitive in our decoupled setup due to the resulting network perturbation. Such perturbation, even when minimal state is being communicated between the modules, prevents

accurate network delay metric usage in a cost function. However, our simplified evaluation design has allowed us to rapidly build prototypes of the fusion and placement modules.

3.8 Related Work

Using a dataflow (task) graph to model distributed applications is a common technique in streaming databases, computer vision, and robotics [22, 83, 71]. DFuse employs a script based interface to express task graph applications over the network similar to SensorWare [14]. SensorWare is a framework for programming sensor networks, but its features are orthogonal to what DFuse provides. Specifically, (*i*) Since DFuse focuses on providing support for fusion in the network, the interface to write fusion-based applications using DFuse is quite simple compared to writing such applications in SensorWare. (*ii*) DFuse provides optimizations like prefetching and support for buffer management which are not yet supported by other frameworks. Other approaches, like TAG [57], look at a sensor network as a distributed database and provide a query-based interface for programming the network. TAG uses an SQL-like query language and provides in-network aggregation support for simple classes of fusion functions. TAG assumes a static mapping of roles to the network, i.e., a routing tree is built based on the network topology and the query on hand.

Data fusion, or in-network aggregation, is a well-known technique in sensor networks. Research experiments have shown that it saves considerable amount of power even for simple fusion functions like finding min, max or average reading of sensors [57, 45]. While these experiments and others have motivated the need for a good role assignment approach, they do not use a dynamic heuristic for the role assignment and their static role assignment approach will not be applicable to streaming media applications.

Research in power-aware routing for mobile ad hoc networks [76, 46] proposes

power-aware metrics for determining routes in wireless ad hoc networks. We use similar metrics to formulate different cost functions for our DFuse placement module. While designing a power-aware routing protocol is not the focus of this work, we are looking into how the routing protocol information can be used to define more flexible cost functions.

3.9 DFuse Conclusions

In summary, I have designed APIs for mapping fusion applications (such as distributed surveillance and environment monitoring) on available resources. I argue that the proposed framework will ease the development of complex fusion applications for distributed networks. The fusion API can be extended in two directions: (*i*) to accommodate more applications, and (*ii*) to allow dynamic reconfiguration of fusion application dataflow graph. Since fusion applications are typically time-sensitive, the fusion API may need to provide QoS specific parameters. Moreover, The fusion application dataflow graph may itself be dynamic with the addition and deletion of nodes to the dataflow graph. The programming APIs need to evolve and take into account these application dynamism.

Although placement module and role assignment algorithms are not my contributions in this dissertation, I have briefly summarized them and highlighted key results. The DFuse framework uses a novel distributed role assignment algorithm that will increase the application runtime by doing power-aware, dynamic role assignment. We validate our arguments by designing a sample application using our framework and evaluating the application on an iPAQ based sensor network testbed.

Developing streaming applications also require run-time mechanisms for high performance computing (HPC) resource allocation for different nodes of a dataflow graph. In the next chapter, I describe my solutions to this second problem, namely, allocating high performance computing resources for data streaming applications to deliver

required quality of service.

CHAPTER IV

STREAMLINE SCHEDULING HEURISTIC

In addition to a uniform programming abstraction (DFuse[52, 69]), developing streaming applications also require run-time mechanisms that allocate resources for different nodes of an application dataflow graph. In this chapter, I address the resource allocation issue for data streaming application.

As mentioned previously, *Streaming applications* have the following characteristics: (*i*) they are continuous in nature, (*ii*) they require efficient transport of data from/to distributed sources/sinks, and (*iii*) they require the efficient use of high-performance computing resources to carry out compute-intensive tasks in a timely manner. The focus of this chapter is addressing the third component of the above characteristics, namely, the use of high-performance computing (HPC) resources to carry out compute-intensive tasks. Consider for example, a video-based surveillance application. The compute intensive part may analyze multiple camera feeds from a region to extract higher level information such as "motion", "presence or absence of a human face", or "presence or absence of any kind of suspicious activity". Such an application is represented as a coarse-grain dataflow graph, wherein the nodes represent increasing sophistication of computations that may need to be performed on the data stream to facilitate the extraction of high-level information.

An interesting aspect of this emerging class of streaming applications is that they are *ubiquitous* and *dynamic*. Thus, HPC resources are needed in a distributed manner to address the dynamic needs of an application. Grid computing offers the ability to harness the ambient HPC resources for a compute intensive problem. Grid computing has its roots in traditional high-performance computing, with initial efforts focused

on scientific and engineering applications. While there have been efforts to expand the reach of the grid to support interactive [78] and streaming applications [21], the scheduling infrastructure available in the grid is largely geared to support batch-oriented applications.

At some level such coarse-grain dataflow graphs resemble task-graphs that have been the focus of multiprocessor scheduling work from the 70's [6, 24, 70, 39, 75, 37, 53, 81]. However, in multiprocessor scheduling, a task-graph (a directed acyclic graph) is used as an ordering mechanism to show the dependencies among the individual tasks of a parallel computation. These dependencies are respected and exploited to create a mapping heuristic (since the scheduling problem is known to be NP-Complete) that maximizes the utilization of the computational resources and reduces the completion time of the application. The coarse-grain dataflow graph of a streaming application, on the other hand, is a representation of the processing that is carried out on the data during its passage through the pipeline of stages. In the steady state, all the stages of the pipeline are working on different snapshots of the stream data. So the scheduling of such streaming applications is not an ordering issue; rather, it is a mapping of the different stages of the pipeline to the available resources respecting the computation and communication requirements of each stage with a view to optimizing the latency and throughput metrics of the entire pipeline. For example in a video-based surveillance application, when the n-th stage is working on the (hitherto transformed results of the) i^{th} frame of video, the first stage of the pipeline is working on the $(n+i)^{th}$ frame. The scheduling of a streaming application on a high-performance computing engine is *data centric*, which is its primary distinguishing characteristic compared to task-graph based scheduling heuristics of yesteryear.

In this chapter, I study the problem of scheduling streaming applications on the grid. The scheduling heuristic, called *Streamline*[8, 9], is designed to adapt to the dynamic nature of grid environment and varying demands of a streaming application.

It runs periodically and takes into account (a) computation and communication requirements of the various stages of the dataflow graph, (b) any application-specified constraints, and (c) current resource (processing and bandwidth) availability. The output of the scheduling heuristic is a placement of the stages of the pipeline on the available HPC resources such that the latency and throughput of the application are optimized. The placement generated by Streamline heuristic can be used by other services (such as task migration) in a grid environment in order to dynamically adapt the performance of an application.

I have designed the scheduling heuristic over the existing grid framework, using Globus Toolkit [35, 34]. To understand the performance of Streamline with respect to an optimal schedule (which is infeasible to implement but for very small applications), I use Simulated Annealing as an approximation for an optimal schedule in our experimental study. I also analyze how existing schedulers in grid can be enhanced to support streaming applications using Condor[79] as example. Most of the existing grid schedulers [19, 12, 79] focus on allocating resources for batch-oriented applications. Condor is a well studied resource allocator for grid and it uses DAGMan[25] for task graph based applications. DAGMan is designed for batch jobs with control-flow dependencies and ensures that jobs are submitted in proper order, whereas different stages of a streaming application work concurrently on a snapshot of data. Thus, I have used Condor scheduler but built additional features to allow instantiation of streaming applications from a batch scheduler. I call this framework as *E-Condor*[1].

I compare the performance of Streamline with Optimal, Simulated Annealing, and E-Condor for "kernels" of streaming applications. The results show that our heuristic performs close (within 1%) to Optimal and Simulated Annealing, and is better than E-Condor by nearly an order of magnitude when there is non-uniform CPU resource

[1]Note: E-Condor does not use Condor's matchmaking capability. We use E-Condor as a demonstration of how a batch scheduler can be used for streaming applications. However, the performance of E-Condor can be improved by using the full functionalities of Condor.

availability, and by a factor of four when there is non-uniform communication resource availability. I consider two variants of Simulated Annealing algorithm with different scheduling overheads and observe that neighbor-selection and annealing schedule in a Simulated Annealing algorithm have a more profound impact for communication-intensive kernels than for computation intensive kernels. My scalability study shows that Streamline is more effective than E-Condor in handling large dataflow graphs, and performs close to Simulated Annealing algorithms, with smaller (by a factor of 1000) scheduling time.

The rest of this chapter is organized as follows. In Section 4.1, I define the scheduling problem. Section 4.2 describes our Streamline scheduling heuristic. I present the overall system architecture that integrates Streamline into the grid computing framework in Section 4.3. The experimental setup, performance evaluation and results are presented in Section 4.4. I explain my work in the context of other related work in Section 4.5 and conclude this chapter in Section 4.6.

4.1 Problem Definition

A scheduling system model in the grid environment consists of an application, available resources, application specific constraints, resource specific constraints, and performance criteria for scheduling. The streaming application is represented by a coarse-grain directed acyclic dataflow graph, $G = (V, E)$, where V is the set of v stages and E is the set of e edges. Each node s_i of the dataflow graph represents a continuously running application stage with the direction of dataflow denoted by the edges. In this application model, each node of the dataflow graph continuously receives data items from the preceding stage, performs computation, and sends data items to the subsequent stages. Each edge $(i,j) \in E$ represents the direction of dataflow such that stage s_j waits for data to arrive from stage s_i before execution. *Ecycle* is a $v \times 1$ matrix of computation data, where $ecycle_i$ is an estimate of the average amount of

CPU cycles required by stage s_i for each streaming data item produced. *Ecomm* is a $v \times v$ matrix of communication, where $ecomm_{i,j}$ is the amount of data required to be transmitted from stage s_i to stage s_j. These processing and communication estimates can be provided by the application for each stage of the dataflow graph, or these estimates can be derived by application profiling as we have done in [86].

Static information (machine architecture, CPU speed, amount of memory, and hardware configuration) about available resources is obtained by querying the information service [26]. Dynamic information (such as estimate of available processing cycles and end to end network bandwidth) is obtained from Network Weather Service (NWS) [88]. Our target computing environment consists of a set Q of q resources. B is a $q \times q$ communication matrix in which $b_{i,j}$ gives an estimate of available network bandwidth between node n_i and node n_j. Similarly, *Proc* is a $q \times 1$ computation matrix in which $proc_i$ gives an estimate of available CPU cycles on node n_i.

Before scheduling, each stage in the dataflow graph is labeled with an average execution cost ($\overline{w_i}$), and each edge is labeled with an average data transmission cost ($\overline{c_{i,j}}$). The average execution cost ($\overline{w_i}$) of stage s_i is measured as a ratio of the required average CPU cycles $ecycle_i$ and average available CPU cycles across the resources. The average data transmission cost for edge (*i,j*), ($\overline{c_{i,j}}$), is defined as the ratio of the estimated data transmission required, $ecomm_{i,j}$, and the average available data transfer across all resource pairs. The scheduler also takes as input a set of application constraints and a set of constraints for available resources. The resource specific constraints are gathered by querying the information service [26]. Each of the constraints specify various site specific policies that may affect the resources allocated to a streaming application.

In a dataflow graph, a stage without any parent is called an *input stage* and a stage without any child is called an *output stage*. After all stages in a dataflow graph are scheduled, the throughput of the application is the actual rate at which data

items are produced by the output stage s_{out}. The *objective function* of the scheduling problem is to determine the assignments of a streaming application dataflow graph to available resources such that the resource and application specific constraints are satisfied and throughput is maximized.

4.2 The Streamline Scheduler

We have developed a grid scheduling algorithm, called *Streamline*, for placement of a streaming application's coarse-grain dataflow graph using available grid resources. Streamline makes the scheduling decision taking into consideration static information of available resources, dynamic information of available processing and communication capabilities in the target environment, and different application and resource specific policies. The scheduling heuristic expects to maximize throughput of the application by assigning the best resources to the most needy stage in terms of computation and communication requirements. Streamline works in three phases: (*i*) *stage prioritization phase* where the stages of the dataflow graphs are prioritized depending on their computation and communication criteria; (*ii*) *resource filtering phase* for filtering available resources based on application and resource specific policies; and (*iii*) *resource selection phase* for selecting the "best" resource that maximizes the throughput of the entire graph. Streamline belongs to the general class of *list scheduling* algorithms [6, 24, 39, 70, 44]. We differ from traditional algorithms in (*i*) stage selection where we take computation and communication into account, (*ii*) estimating the required computation and communication cost for a stage in the dataflow graph, (*iii*) taking into account the resource and application specific policies, and (*iv*) taking into consideration the dynamic information about available processing and communication capabilities in the target environment.

Stage Prioritization Phase: This phase considers the computation and communication cost of the dataflow graph in assigning priorities to different stages. The

computation and communication intensive tasks get higher priorities over the other tasks. Also, the remaining execution time to process a particular data item by all subsequent stages is taken into account where a stage with the highest cost path to the output stage gets priority.

Two terms *rank* and *blevel* are introduced here. *Rank* calculates the average computation and communication cost of a stage and *blevel* estimates the overall remaining execution time of a data item after being processed by a stage. The *blevel* of a stage s_i is the cost of the longest path from from s_i to an exit node and is recursively defined by

$$blevel(s_i) = \overline{w_i} + \max_{s_j \in succ(s_i)} (\overline{c_{i,j}} + blevel(s_j)) \quad (1)$$

where *succ(s_i)* is the set of immediate successors of stage s_i, $\overline{w_i}$ is the average computation cost of stage s_i and $\overline{c_{i,j}}$ is the average communication cost of edge (*i,j*).

The *rank* of a stage is the sum of its communication and computation cost, and gives a rough estimate of the total computation and communication time required by the stage to fetch all input data items, process them, and send the result to successive stages. The *rank* is used in relative ordering of the stages based on their requirements *before* an actual assignment is made. Therefore, this simple model suffices in giving higher *rank* to a more needy stage. We assign a *rank* to each stage s_i, taking into account the average computation and communication cost as

$$rank(s_i) = \overline{w_i} + \sum_{s_j \in pred(s_i)} \overline{c_{j,i}} + \sum_{s_k \in succ(s_i)} \overline{c_{i,k}} \quad (2)$$

where *pred(s_i)* is set of immediate predecessors of stage s_i.

The priorities are assigned in the following order: (*i*) The stage with a higher *rank*, as calculated by Equation 2, gets a higher priority, (*ii*) For two stages with same *rank*, we assign higher priority to the stage with higher *blevel*, thereby giving preference to a stage along a higher cost path to the output stage, (*iii*) For stages with same *rank* and *blevel*, tie-breaking is done randomly. The stages are considered in priority order

and are allocated the "best" available resources.

Table 2: Streamline Scheduling Algorithm

// Input: dataflow graph G(S,E), resource set R
// Output: assign stages $s_i \in$ S *to resources* $n_j \in$ R
/ Initialize the dataflow graph */*
for (stage $s_i \in S$, edge $e_{i,j} \in E$ in dataflow graph G(S,E))
  $s_i = w_i$ *//*w_i: avg execution cost
  $e_{i,j} = c_{i,j}$ *//*$c_{i,j}$: avg transmission cost

/ Set priority to stages */*
for (every stage $s_i$in the dataflow graph G(S,E))
  blevel$(s_i) = \overline{w_i} + \max_{s_j \in succ(s_i)}(\overline{c_{i,j}} + \text{blevel}(s_j))$
  rank $(s_i) = \overline{w_i} + \sum_{s_j \in pred(s_i)} \overline{c_{j,i}} + \sum_{s_k \in succ(s_i)} \overline{c_{i,k}}$

Sort stages in S into list SLIST in decreasing order of *rank*, use *blevel* to break ties

/ Choose the best resource */*
while (there are unscheduled stages in SLIST)
  Select the first stage s_i from SLIST

  / resource filtering using application and resource policies */*
  Construct candidate set R by filtering available resources based on policies

  / compute cost of assignment (A) */*
  for (each resource $n_k \in R$)
    $A(n_k, s_i) = ecycles_i \,/\, proc_j + \sum_{s_k \in pred(s_i)} ecomm_{k,i} / \overline{b_{in}(j)} + \sum_{s_k \in succ(s_i)} ecomm_{i,k} / \overline{b_{out}(j)}$

  / pick the best assignment for the stage */*
  Output (s_i, n_k) pair with min$(A(n_k, s_i))$

Resource Filtering Phase: In this phase of the scheduling algorithm, we filter out available resources that may not be permissible by the application or resource policies and obtain a set R of r candidate resources. We take into account the application specific static resource requirements as well as resource specific policies. Even though there may be a large number of available resources, the candidate resource set may be small after this step, depending on how restrictive the application and resource

policies are. Some examples of application specific constraints are (*i*) collocating stages of a streaming application on the same resource to reduce communication latency, (*ii*) any special requirements of a stage such as a graphics co-processor, (*iii*) QoS requirements specifying desired throughput and latency, and (*iv*) application defined priorities among different choices such as image and audio quality degradation for application adaptation.

Resource Selection Phase: In this phase the appropriate resources are picked from the set of candidate resources obtained in the resources filtering phase. Unlike most of the task graph schedulers that take only the computation capability to select resources, Streamline considers available CPU as well as end-to-end bandwidth. Streamline evaluates a particular resource using a cost function that computes the cost of assigning a stage to a resource node. The cost function estimates the computation and communication time for a particular assignment of a stage and picks a resource which gives the least cumulative time. Since the stages are considered in the order of their resource requirements (represented by *rank*) which may not be consistent with the dataflow order, Streamline uses estimates of average input and output bandwidth availability for each resource in calculating the cost of an assignment. By using information gathered from NWS [88], Streamline algorithm estimates the average incoming ($\overline{b_{in}(i)}$) and average outgoing bandwidth ($\overline{b_{out}(i)}$) for each resource node n_i as follows:

$$\overline{b_{in}(i)} = \sum_{n_j \in R} b_{j,i}/r \tag{3}$$

$$\overline{b_{out}(i)} = \sum_{n_k \in R} b_{i,k}/r \tag{4}$$

where R is the candidate resource set of r resources and $b_{i,j}$ is an estimate of available end to end network bandwidth between node n_i and node n_j.

The cost ($A\ (n_j,\ s_i)$) of allocating resource node $n_j \in R$ to stage s_i is computed by summing up the estimated computation and communication cost for this particular

assignment and is represented as

$$A(n_j, s_i) = ecycles_i/proc_j + \sum_{s_k \in pred(s_i)} ecomm_{k,i}/\overline{b_{in}(j)} + \sum_{s_k \in succ(s_i)} ecomm_{i,k}/\overline{b_{out}(j)}$$

where $ecycle_i$ is an estimate of the average amount of CPU cycles required by stage s_i, $proc_j$ is an estimate of available CPU cycles on node n_j and $ecomm_{i,k}$ is the average amount of data transferred from stage s_i to stage s_k. To each stage s_i of the dataflow graph, we assign a resource n_j that has the minimum cost ($A\ (n_j,\ s_i)$) as calculated by Equation 5. Since we consider end-to-end network bandwidth available between each pair of resources in the candidate set, the Streamline algorithm has $O(v \times r^2)$ time complexity where v is the number of stages in the dataflow graph and r is the number of resources in the candidate set R. Even in the presence of a large number of resources, we expect the candidate set for each stage to be small and the time complexity to be admissible. The complete algorithm is presented in Table 4.2.

To eliminate any ambiguity, the algorithm also takes into account the following points: (*i*) When multiple stages get assigned to the same resource, distribute the available bandwidth and CPU resources equally between all the stages in calculating the cost of the assignment[2]. (*ii*) In case there are multiple remaining candidate nodes, pick a node among the remaining candidate nodes at random. By randomly picking a node, the scheduler expects to distribute load among the equally desirable resources for a particular stage.

Since our scheduling heuristic works by picking best resource for each stage of the dataflow graph, additional policies concerning resources, applications, and local schedulers can be easily incorporated in calculating the cost of a particular assignment. In the next section, we present our system architecture that integrates the Streamline scheduling heuristic into grid computing framework.

[2]Assuming that the local scheduler equally allocates the available CPU and network bandwidth, and that all the stages contend for CPU and network usage simultaneously. Given any additional information, the cost calculation can be accordingly adjusted.

4.3 System Architecture

We have designed a system that enables resource allocation for streaming application using grid. Our system architecture is guided by the design goals of making best use of existing grid functionalities and operating in the presence of dynamic resource availability and node connectivity by taking into account non-uniform resource characteristics.

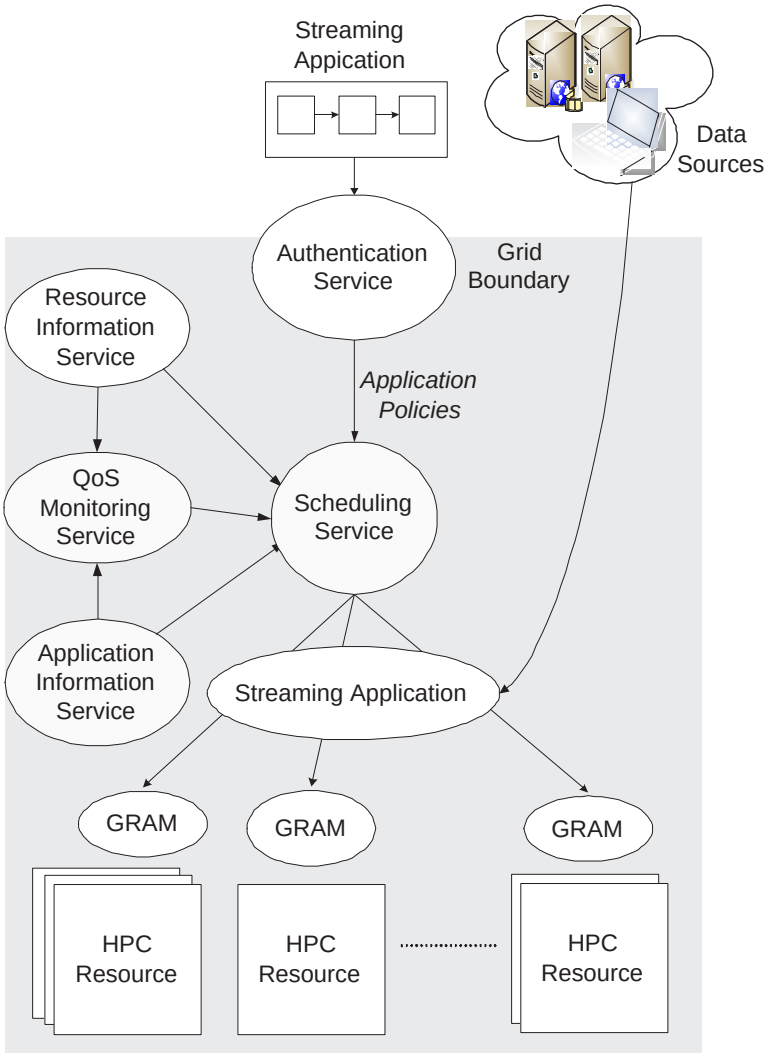


Figure 15: Resource Allocation System Architecture

Figure 15 shows the system which uses the existing grid functionalities of Globus

Toolkit[35], the Network Weather Service[88] for current information and future prediction of the resources, and some additional services introduced to make the streaming scheduler function properly.

The services that are part of the present Globus Toolkit infrastructure are (*i*) *Authentication Service* used for authentication to grid resources using Grid Security Infrastructure (GSI); (*ii*) *Resource Information Service*[26] used for obtaining information about the grid resources; and (*iii*) *Grid Resource Allocation and Management(GRAM)*[27] service used for submitting jobs to the grid. *Network Weather Service* provides dynamic information (such as CPU usage and end to end bandwidth) about available resources. Among the services we have integrated to the grid include *Application Information Service* to keep track of the streaming application status and the *QoS Monitoring service* to checks if the desired QoS is met. The *Scheduling Service* makes all the scheduling decisions by contacting the QoS monitoring service, Network Weather Service, and Resource Information Service. It runs the scheduling algorithm periodically to make sure the proper assignment of resources is done and if the QoS is violated beyond a threshold, a reallocation process takes place. The forecasts from the Network Weather Service are used in making task migration decisions. The details are out of the scope of this thesis. Our focus is on the scheduling heuristic in this work.

The whole system works as follows. The user authenticates to the grid using the Authentication Service. Then the application is submitted as a coarse-grain dataflow graph to the scheduling service. The scheduling service contacts Resource Information Service [26, 88] and Application Information Service and allocates resources. After resource allocation, the application dataflow graph is instantiated on individual resources using Grid Resource Allocation and Management(GRAM) [27] system. The application then accesses the data from data sources directly. The QoS Monitoring Service monitors the quality of service requirements of the application and

dynamically adapts resource assignment by contacting the Scheduling Service. The QoS monitoring service infers the computational requirements of the running application by periodically contacting the Application Information service and the Resource Information Service, and provides this information to the scheduler.

4.4 Performance Evaluation

In this section, we study the performance of the Streamline scheduler for supporting streaming applications. Streamline provides a rich set of facilities that include: (*i*) APIs for the user to specify the resource requirements of each stage and the dependencies among the stages of the streaming application; (*ii*) APIs for job submission that allow multiple applications to be submitted to the scheduler at the same time; each application receives a distinguished name; and (*iii*) APIs for querying job status using the distinguished name.

For evaluating the performance of Streamline, we take the following approach. First, we develop a baseline scheduler for streaming applications using Condor, a well-accepted grid scheduler for batch-oriented applications. The resulting framework, called *E-Condor*, uses Condor to obtain the resources necessary to launch the individual stages of a streaming application. But prior to launching, E-Condor takes care of setting up all the necessary coupling between the stages commensurate with the dataflow graph of the application. Further, to understand the performance of Streamline with respect to an optimal schedule (which is infeasible to implement but for very small applications), we use Simulated Annealing as an approximation for an optimal schedule in our experimental study.

Thus our experimental study has four parts: (*i*) We present an Optimal placement (for small dataflow graphs) and approximation algorithms using Simulated Annealing (for both small and large dataflow graphs) for comparison with Streamline; (*ii*) We

investigate how existing grid schedulers can be used to deal with streaming applications; this has resulted in the baseline framework called E-Condor; (*iii*) We compare the performance of Streamline with Optimal placement, two variants of Simulated Annealing (SA1 and SA2), and E-Condor for executing "kernels" of streaming applications; (*iv*) We evaluate the scalability of Streamline with respect to Optimal, Simulated Annealing, and E-Condor approaches.

4.4.1 Optimal Placement Algorithm

The Optimal placement algorithm explores all possible assignments of resources to the individual stages of a dataflow graph and selects an assignment with the minimum cost. The cost of an assignment represents an estimate of the time it takes to produce a single output item as data is processed by the various stages of a dataflow graph. Since we are interested in observing the relative costs of different assignments, we use a simple model where this cost (F) is estimated by summing up the estimated computation and communication time of each stage of the dataflow graph as follows:

$$F([n_0...n_v],[s_0...s_v]) = \sum_{i=0}^{i=v}(ecycles_i/proc_i) + \sum_{(i,j)\in E}(ecomm_{i,j}/b_{i,j}) \quad (5)$$

where resource node n_i is allocated to stage s_i in the assignment under consideration, and $(i,j) \in E$ represents an edge between stages s_i and s_j in the dataflow graph.

The above cost model makes a conservative assessment by summing up the estimate of individual computation and communication times, ignoring any parallelism. The cost model also ignores networking and buffer management overheads in transmission. However, since we are interested only in the relative ordering of different assignments, use of such a simple cost model is justified. For v stages and r candidate resources, the computational complexity of the Optimal algorithm is approximately equal to the number of permutations of v resources out of r (^rP_v). Thus, Optimal algorithm is computationally infeasible for large dataflow graphs.

4.4.2 Simulated Annealing Algorithms (SA1, SA2)

For large dataflow graphs, computing an optimal schedule is infeasible. Hence we use Simulated Annealing as an approximation to Optimal for comparison purposes. Simulated Annealing is computationally intensive and in general not the best candidate (from the point of scheduling overhead) to use as a scheduling heuristic. The intent is to show that Streamline heuristic yields a schedule that is close in performance to an exhaustive search technique (such as Optimal or SA), while being computationally feasible in terms of scheduling overhead.

Simulated Annealing [60, 51] is a generalization of a Monte Carlo method for statistically finding a global optimum for a multivariate function. It uses the concept of repeated heating and slowly cooling (*annealing* process) a crystalline structure to bring it to a more ordered state and has been used in Operation Research to successfully solve a variety of optimization problems [51]. In simulated annealing, a system is initialized at *temperature* T with some configuration with cost F_0. New configuration is constructed by applying a random perturbation and change in cost dF is computed - if the new configuration lowers the cost of the system, it is unconditionally accepted otherwise it is accepted with a probability given by the Boltzmann factor $\exp(-dF/T)$ [51]. This processes is repeated sufficient times at the current temperature to sample the search space (by visiting *neighbors* of the current configuration). Then, the temperature is decreased as specified by a chosen *annealing schedule* and the entire process is repeated at successive lower temperature until a terminating condition (frozen state) is reached. This procedure allows the system to move to a lower cost state, while still getting out of local minima due to probabilistic acceptance of some upward moves.

The state space of our scheduling algorithm is all possible assignments of candidate resources to the stages of a dataflow graph. We use the same cost function as in the Optimal algorithm (Equation 5) and $\exp(-dF/T)$ as the transition probability. Since

overheads of the scheduling algorithm is also critical in our problem, we have selected two different strategies for neighbor selection and annealing schedule (*SA1* and *SA2*). SA1 has run-time complexity comparable to Streamline whereas SA2 requires longer running time, thereby expecting to produce better schedules. At a fixed temperature, SA1 considers v randomly selected neighbors whereas SA2 considers v^2 neighbors for a v stage dataflow graph. SA1 and SA2 also differ in the length of the annealing schedule whereby SA1 considers at most r^2 temperature reductions. We have also employed a simple optimization heuristic that avoids repetitive transitions between the same two states at a fixed temperature by considering the neighbors in a fixed order. The details of the algorithms are presented in Table 3.

4.4.3 Using a Batch Scheduler for Streaming Applications

Most of the existing grid schedulers [19, 12, 79] focus on allocating resources for batch-oriented applications. We have selected Condor [79], due to its maturity and flexibility, as a vehicle for comparison of an existing grid scheduler against Streamline.

Condor uses DAGMan[25] to launch applications that are specified by a task-graph. DAGMan is designed for task-graph based batch jobs with control-flow dependencies and hence launches a stage s_i of a task-graph only after all stages $s_j \in$ *pred(s_i)* have *finished* execution. However, as observed before, in a streaming application, each stage of the dataflow graph is concurrently working on a snapshot of the continuous stream data. Therefore, we have developed a simple stream scheduling framework, called *E-Condor*. E-Condor uses Condor to obtain the resources necessary to launch the individual stages of a streaming application. But prior to launching, E-Condor takes care of setting up all the necessary coupling between the stages commensurate with the dataflow graph of the application. E-Condor is not aware of resource requirements of different stages of the dataflow graph, it allocates resources to the stages in the order in which they are submitted using Condor scheduler.

Table 3: Simulated Annealing Algorithms (SA1 and SA2)

// Input: dataflow graph G(S,E), resource set R
// Output: assign stages $s_i \in$ S *to resources* $n_j \in$ R

/ Initialize the Simulated Annealing */*
Select v random resources from r eligible machines
Compute initial cost of assignment (F_0):
$F_0([n_0...n_v],[s_0...s_v]) = \sum_{i=0}^{i=v}(ecycles_i/proc_i) + \sum_{(i,j)\in E}(ecomm_{i,j}/b_{i,j})$
Compute average increase in cost across neighbors ($\overline{dF_0}$)
$X_0 = 0.8$ // average increase acceptance probability
$T_0 = -\overline{dF_0}/\ln(X_0)$ //starting temperature

/* Annealing Process */
repeat {at each temperature}
/* Neighbor selection for SA1 and SA2 */
(**SA1:**) **for** v steps **do**
(**SA2:**) **for** v^2 steps **do**
/* perturb the schedule randomly */
swap assignments for two allocated machines or
replace an allocated machine with a free machine
compute change in cost (dF)
if dF is negative, accept new schedule unconditionally
else accept if a random number $< \exp(-dF/T)$
decrease temperature by a factor of α (0.95)
/* terminating conditions for SA1 and SA2 */
(**SA1:**) **until** r^2 steps or temperature is above threshold (0.001) or
cost does not change
(**SA2:**) **until** temperature is above threshold (0.001) or
cost does not change

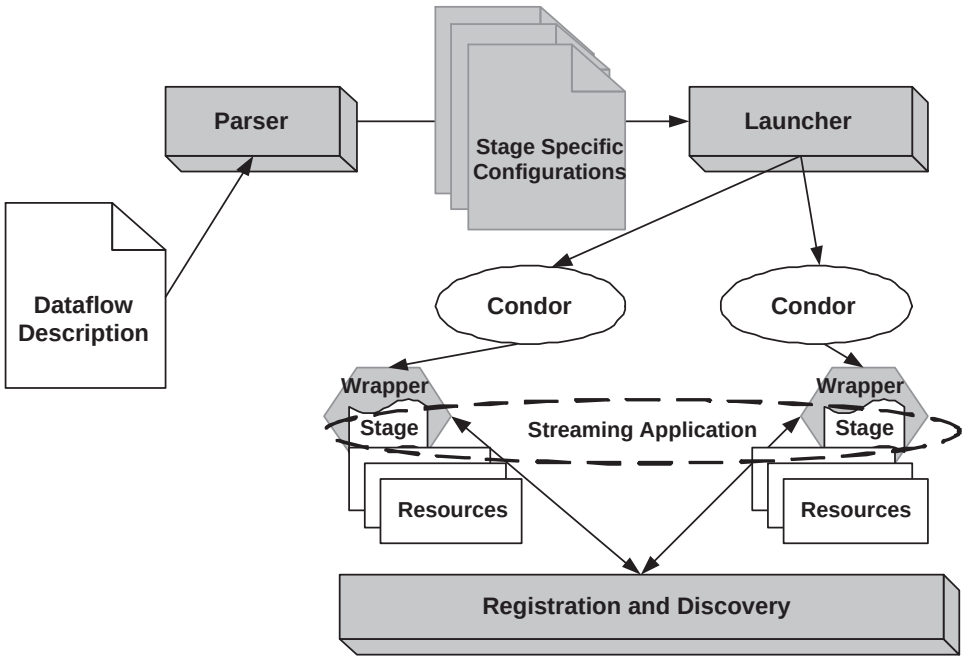


Figure 16: E-Condor Architecture

E-Condor architecture, shown in Figure 16, has four components: (*i*) A *parser* that automatically generates the entire dataflow graph and the per-stage configuration files given a high-level description of the streaming application; (*ii*) A *launcher* that uses Condor to map the stages of the dataflow graph to different computational nodes provided by Condor; (*iii*) A *registration and discovery* service for establishing the predecessor/successor relationships among the stages of the dataflow graph after they are launched. The architecture automatically generates wrapper code for each stage to register itself with this service, determine its predecessors and successors using the per-stage configuration file, and establish the necessary connections to them; and (*iv*) A *synchronization protocol* that ensures that all the stages have established the necessary connections to one another before actually starting the application-supplied code for that stage.

4.4.4 Distributed Surveillance Application

We use a mock-up of a distributed video-based surveillance application as an example streaming application for the performance study. To arrive at a realistic model of the pipeline that represents this application, we use the following representative image manipulation functions that may form part of this hierarchical processing pipeline: (*i*) **Collage**: A simple concatenation of two images to produce a composite output; (*ii*) **EdgeDetect**: An algorithm to determine the boundaries of objects in an image; (*iii*) **MotionDetect**: An algorithm that computes the magnitude and centroid of inter-frame differences in images to derive inferences on motion; and (*iv*) **FD/FR**: A compute-intensive algorithm to detect and recognize faces in an image that is based on skin tone analysis.

For each of these functions, we use the computation and communication numbers reported in [86], summarized in Table 4. These numbers are the result of profiling these functions on a StrongARM SA-1110 processor. We construct a pipeline consisting of these functions to serve as the workload for our scheduling experiments [3].

Table 4: Computation, Communication Costs of Basic Image Processing Functions

	CPU Cycles	Datasize(Bytes) (I/0)
Collage	803.4K	112K/112K
EdgeD	2616.2K	56K/56K
MotionD	1009K	56K/56K
FD/FR	1959M	30K/30K

[3]Note: Although our experimental setup (see Section 4.4.5) uses an x-86 cluster, the use of these numbers is justified since we are only interested in relative performance of the different schedulers on the same dataflow graph.

4.4.5 Modeling Resource Contention

We define four control variables to model the contention and non-uniformity of resource availability as follows: (*i*) *Mean processing availability* (μ_p) is the the average CPU cycles available across all the nodes normalized by the maximum CPU availability so that this is a number between 0 and 1; (*ii*) *Mean network bandwidth availability* (μ_{bw}) is the average end-to-end network bandwidth available across all pairs of nodes normalized by the maximum network bandwidth availability between any two nodes so that this is a number between 0 and 1; (*iii*) *Variance in processing availability* (σ_p^2) is the variance in CPU cycle availability across all the nodes; and (*iv*) *Variance in network bandwidth availability* (σ_{bw}^2) is the variance in end-to-end network bandwidth availability across all pairs of nodes.

μ_p and μ_{bw} are indicators of the amount of resource contention in the system. σ_p^2 and σ_{bw}^2 indicate the non-uniformity of the load distribution in the system, with higher variance implying higher non-uniformity in load characteristics of the resources. Clearly, the possible sets of values for these four control variables are quite large. Thus, to keep the scope of the experimental study manageable, we study the performance for a chosen subset of values. Further, to keep the discussion simple as well as to understand the effects of these control variables better, we separately study CPU and network contention.

In the experimental study, we fix three of the control variables and study the effect of the fourth on the performance. We assign the resource settings (i.e., CPU and network bandwidth) to correspond to the desired value of the control variable. We experiment with 3 different values for fixed variables in each study and 3 resource assignments for each combination of control variables. For each experiment, we pick 7 data points for the control variable under study with multiple runs corresponding to each data point. Because of our controlled settings, we observed variance across different runs to be negligible ($< 0.01\%$). We present a representative subset of the

results to keep the presentation within limits.

Our experimental platform consists of a sixteen node cluster with dual gigabit-Ethernet interconnects. Each node consists of eight Pentium III 550MHz Xeon processors with 4GB RAM. To model resource contention in a controlled manner in our experiments we adopted the following strategy. We introduce a synthetic delay in the code for a stage commensurate with the (assumed) load on the node that it is running on (parameterized by the assigned setting for that node). This strategy helps simulate non-uniform processor bandwidth availability in a controlled manner. Similarly, in order to model non-uniform network bandwidth availability, we inflate the data size of the data communication between stages in proportion to the level of (assumed) network contention (once again parameterized by the assigned setting).

4.4.6 Micro Measurements

For the micro measurements, we use 4 nodes of the cluster (one processor in each node). We use two "kernels" of a distributed surveillance application in these measurements. There are two sets of measurements, one for a compute-bound kernel and the other for a communication-bound kernel.

4.4.6.1 Compute-Bound Kernel

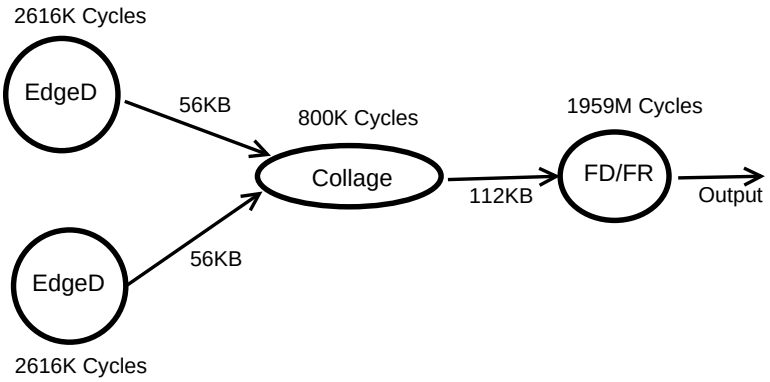


Figure 17: Compute-bound Kernel

Figure 17 shows the compute-bound kernel. The scheduler maps the 4 stages of

the pipeline to the 4 nodes of the cluster. As mentioned earlier, we assign the settings (CPU load and network contention) on the processor and the links commensurate with the control variable value for a particular experiment. The metric for comparison of different scheduling algorithms is the average time taken per output data item averaged over 100 data items.

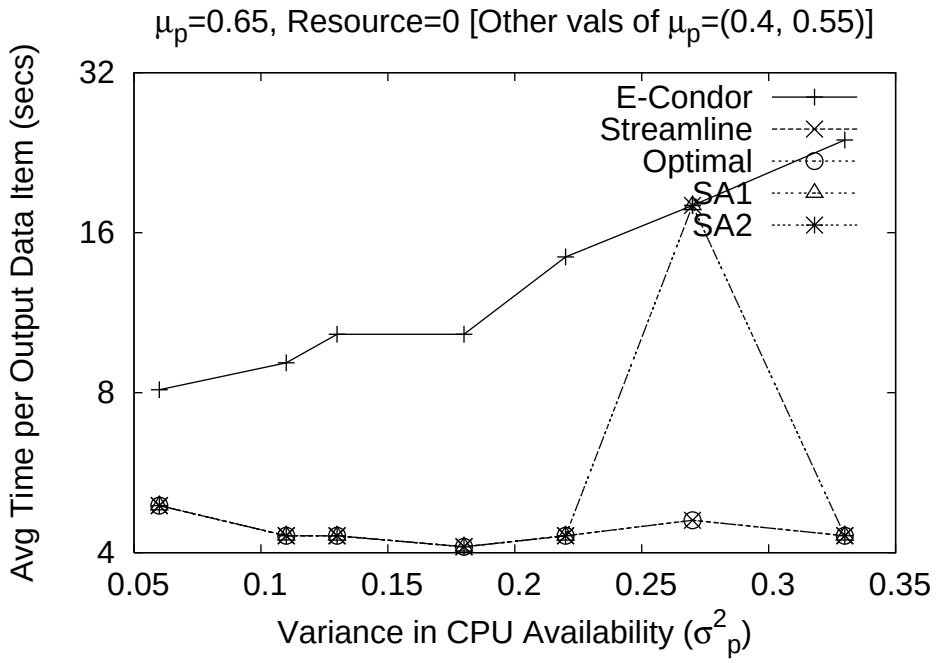


Figure 18: Effect of CPU Availability Variance on Compute Bound Kernel (μ_p=0.65, σ^2_{bw}=0, μ_{bw}=1)

Control variable: CPU Variance. Figure 18 shows the performance of different schedulers when CPU load distribution is non-uniform. It can be concluded from the graph that: (*i*) Performance of Optimal and Simulated Annealing algorithms (SA1, SA2) are comparable, except in one instance, (*ii*) Streamline performs close (within 1%) to Optimal and nearly an order of magnitude better than E-Condor under highly non-even load distribution, and (*iii*) for a given system load (μ_p) the performance of Streamline (like Optimal and Simulated Annealing) *improves* with increasing variance whereas that of E-Condor degrades. E-Condor does not consider

the resource requirements of each stage of a dataflow graph but simply allocates resources to the stages in the order the requests are submitted. On the other hand, Streamline is able to match the variance in the computational requirements of the stages in the application dataflow graph with the variance in the system load to get a better mapping of the stages to the resources. We have observed similar results for other values of μ_p (0.4, 0.55) and 3 resource configurations for each particular combination of control variables. The degradation in performance of SA1 in one instance is attributed to its neighbor selection policy, though we have observed that SA1 performs close to SA2 in other configurations for this experiment.

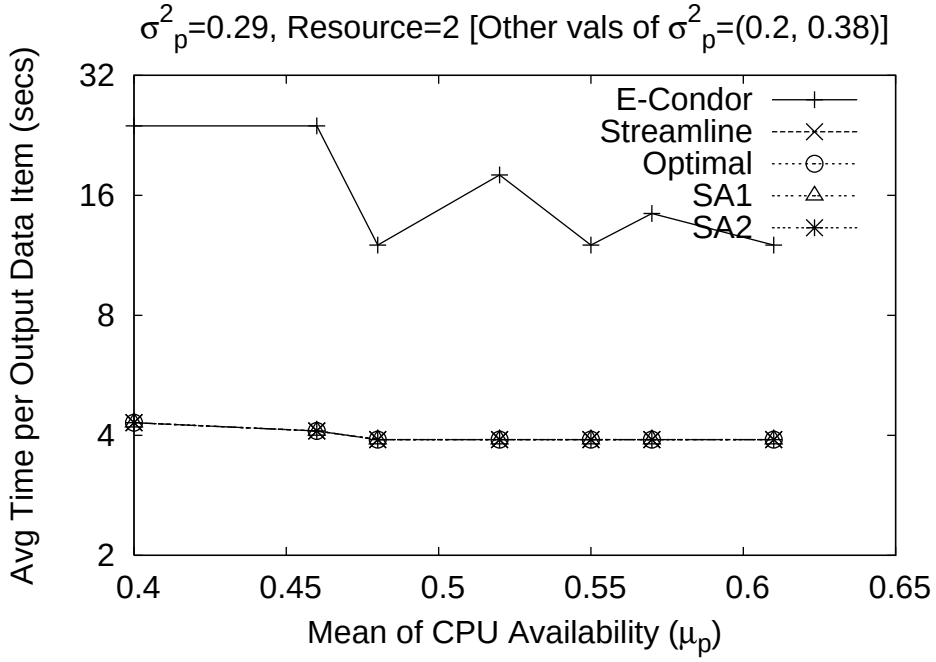


Figure 19: Effect of Mean CPU Availability on Compute Bound Kernel (σ_p^2=0.29, σ_{bw}^2=0, μ_{bw}=1)

Control variable: CPU Availability. Figure 19 shows the effect of mean CPU availability for a fixed variance. The results show that Streamline, Optimal, and Simulated Annealing algorithms are comparable (within 1%) in performance. We also observe that with an increase in mean CPU availability, the performance of

both E-Condor and Streamline improves. However, we see that like Optimal and Simulated Annealing, Streamline does not benefit as much with an increase in mean CPU availability as E-Condor does in some cases. The reason is quite intuitive. Streamline takes advantage of the variance in resource availability in its heuristic when it allocates "best" resource to the most needy stage as determined after stage prioritization; therefore, small increases in the mean CPU availability have little impact in its placement decision and hence in the overall performance. E-Condor, on the other hand, due to its first-fit approach, incurs a severe performance penalty when the mean CPU availability is low. Similar results have been observed for other values of σ_p^2 (0.2, 0.38) and 3 resource configurations for each combination of control variables.

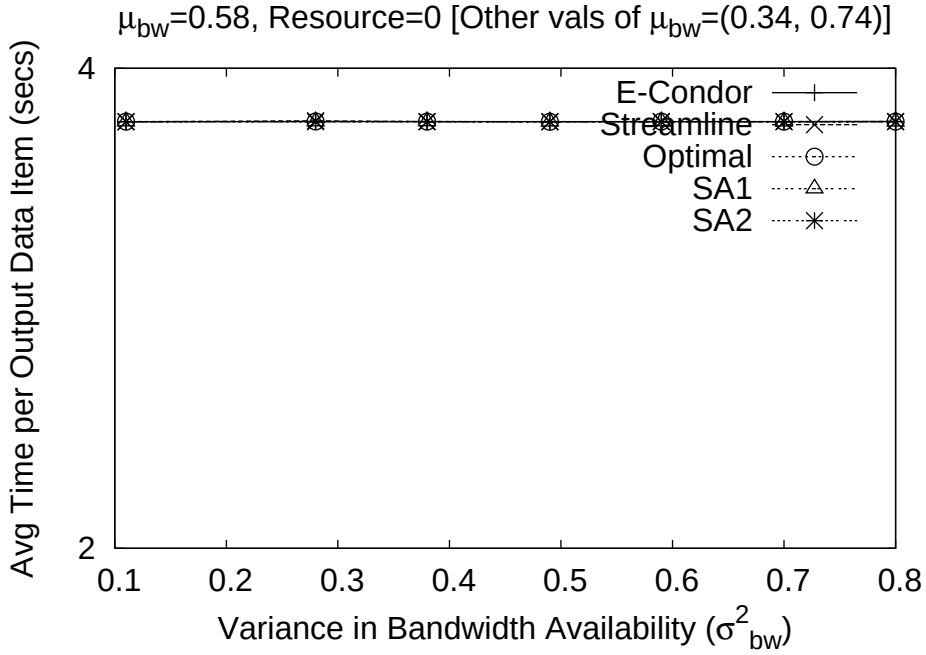


Figure 20: Effect of Variance in Bandwidth Availability on Compute Bound Kernel (μ_{bw}=0.58, σ_p^2=0, μ_p=1)

Control variable: Network Bandwidth Variance. For completeness, we also measure the effect of variance in available bandwidth on the compute bound kernel.

As the results in Figure 20 show, none of the algorithms are affected by the variation in available bandwidth, thereby confirming the computational nature of this kernel. We have observed similar results for other values of μ_{bw} (0.34, 0.74) and all 3 resource configurations for each combination of control variables. We have also conducted experiments studying the effect of network bandwidth availability for 3 different fixed values of σ^2_{bw} (0.32, 0.58, 0.76). These results show (not presented here) the performance of all algorithms to be equivalent for the compute bound kernel, as expected.

In summary, we observe that for the compute bound kernel, performance of Streamline is close (within 1%) to Optimal and SA2, and an order of magnitude better than E-Condor. Moreover, the performance of SA1 is close to Optimal in most cases. This implies that for the compute bound kernel, even a Simulated Annealing algorithm that examines fewer states (SA1) performs close to Optimal. The reason is that for the compute bound kernel, the performance of a particular stage is independent of the placement of other stages. Therefore, even a simple random neighbor selection policy performs well.

4.4.6.2 Communication-Bound Kernel

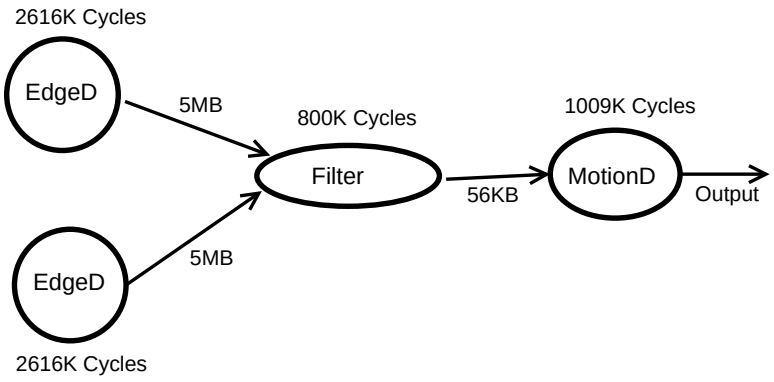


Figure 21: Communication-bound Kernel

Figure 21 shows the communication-bound kernel used for this set of micro measurements. In this kernel, a synthetic *Filter* function is used. The filter function

receives large amounts of data (video images plus their boundaries) from two *edge detectors*. The filter function selects the subset of the input to send on to a *motion detector* for higher level inference.

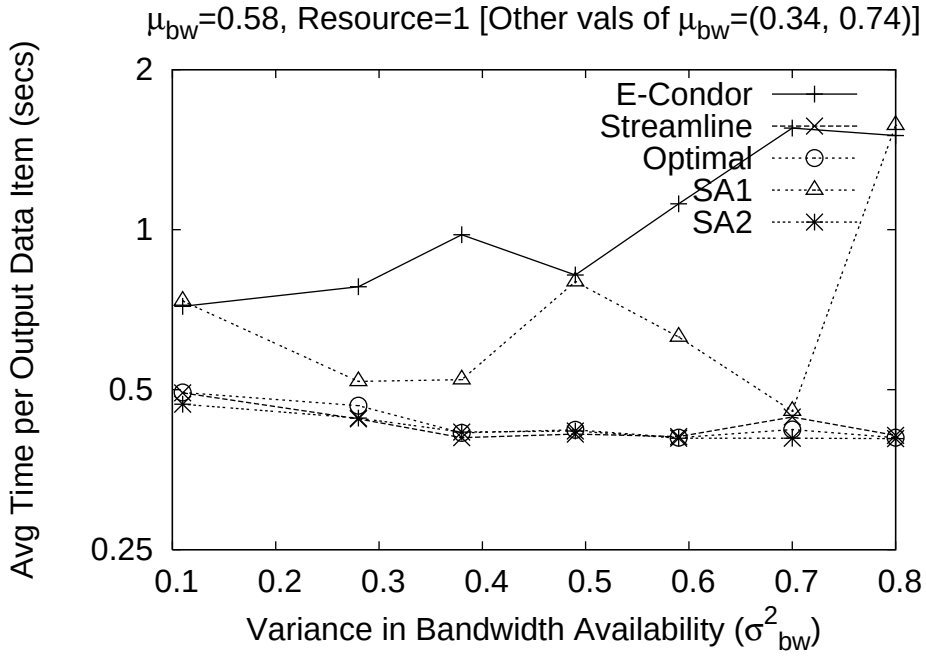


Figure 22: Effect of Bandwidth Availability Variance on Communication Bound Kernel (μ_p= 1, σ_p^2=0, μ_{bw}=0.58)

Control variable: Network Bandwidth Variance. Figure 22 shows the effect of non-uniform bandwidth for the five schedulers. We observe that performance of SA1 is in between E-Condor and Optimal. The performance difference between SA1 and SA2 shows that for a communication bound kernel, a Simulated Annealing algorithm needs to explore a relatively larger part of the search space in order to deliver results comparable to Optimal. In contrast, Streamline heuristic performs close (within 1%) to Optimal and better than SA1. In addition, Streamline out-performs E-Condor by a factor of four under high variance. We also note that Streamline's performance improves under non-uniform load condition due to efficient placement of stages. We have observed similar results for other values of μ_{bw} (0.34, 0.74) and all 3 resource

configurations for each combination of control variables.

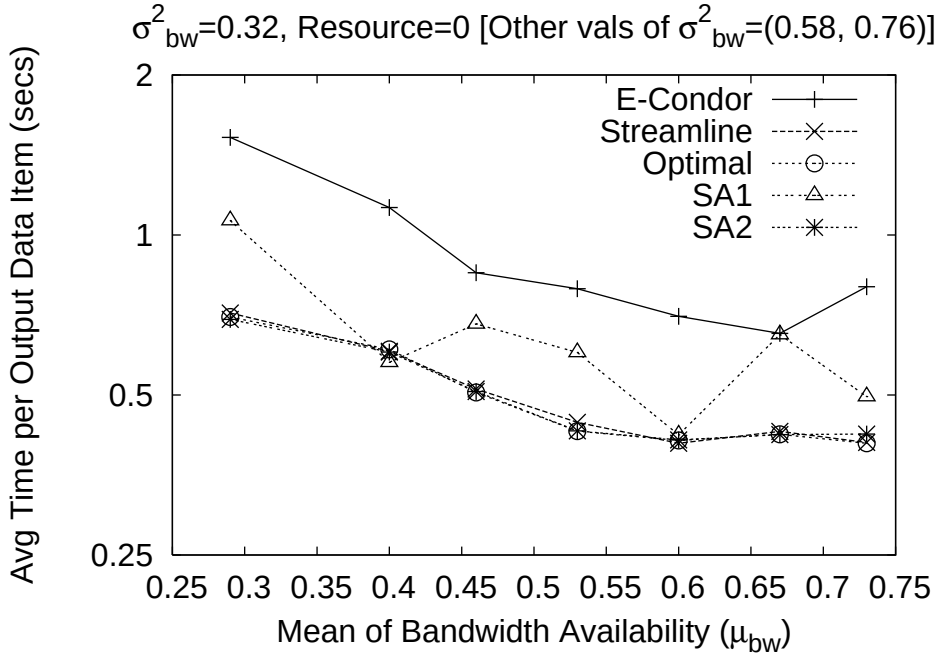


Figure 23: Effect of Mean Bandwidth Availability on Communication Bound Kernel (σ_p^2=0, μ_p=1, σ_{bw}^2=0.32)

Control variable: Network Bandwidth Availability. Figure 23 shows the effect of network bandwidth availability on the performance of the five schedulers. The performance of all the algorithms improve in general as mean bandwidth increases. However, we observe that SA1 performs worse than SA2 in many cases signifying the importance of neighbor selection and length of annealing schedule of a Simulated Annealing algorithm, particularly for communication intensive dataflow graphs. Streamline, in contrast, performs close (within 1%) to Optimal and SA2 in all instances. The performance advantage of Streamline is attributed to the heuristic algorithm in which we take into account computation and communication requirements of different stages as well as dynamic resource availability. We have observed similar results for other values of σ_{bw}^2 (0.58, 0.76) and 3 resource configurations for each combination of control variables.

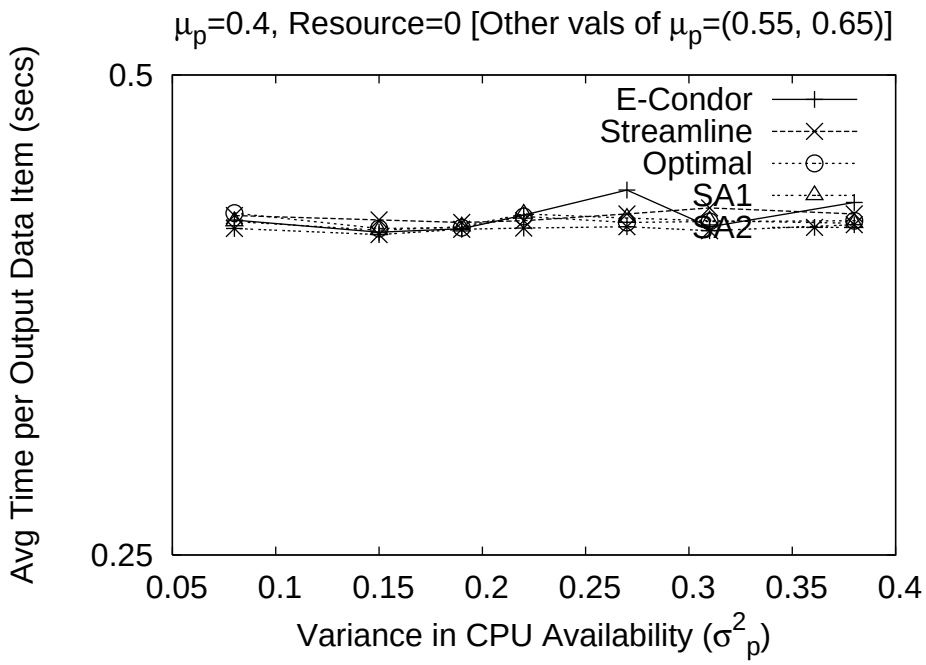


Figure 24: Effect of CPU Availability Variance on Communication Bound Kernel ($\mu_p = 0.4$, σ^2_{bw}=0, μ_{bw}=1)

Control variable: CPU Variance. Figure 24 shows that the communication bound kernel gives similar performance under varying computational load conditions for all the algorithms, thus confirming its communication intensive nature. We have observed similar results for other values of μ_p (0.55, 0.65) and 3 resource configurations for each combination of control variables. We have also noticed that CPU availability has little effect on the relative performance of all the algorithms for the communication bound kernel (graphs not presented here).

Through these micro measurements we establish that Streamline performs significantly better than E-Condor in general, and especially under non-uniform load conditions. Moreover, performance of Streamline is comparable (within 1%) to the Optimal and SA2 algorithms. We also establish that for a communication bound kernel, SA2 performance is better than SA1, thereby illustrating the relative importance of *neighbor selection* and *annealing schedule* strategies for communication intensive dataflow graphs.

4.4.7 Scalability

For the scalability study, we consider a video-based tracking application where multiple camera feeds are analyzed to identify any suspicious activity.

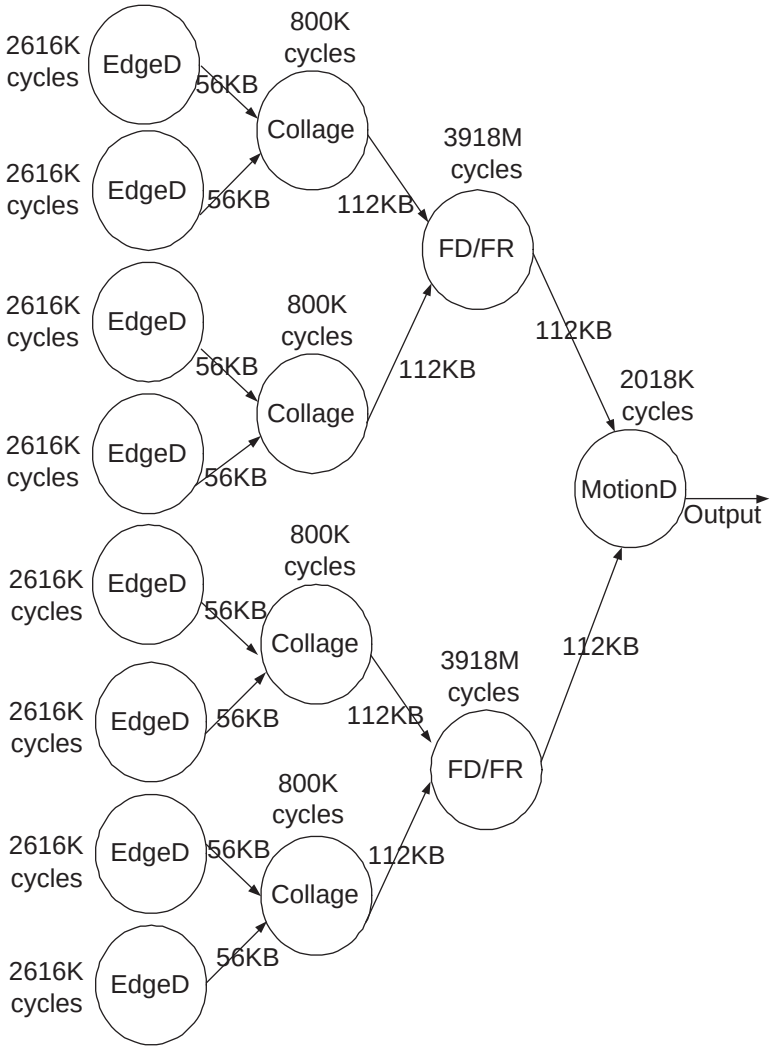


Figure 25: Video-based Tracking Dataflow Graph

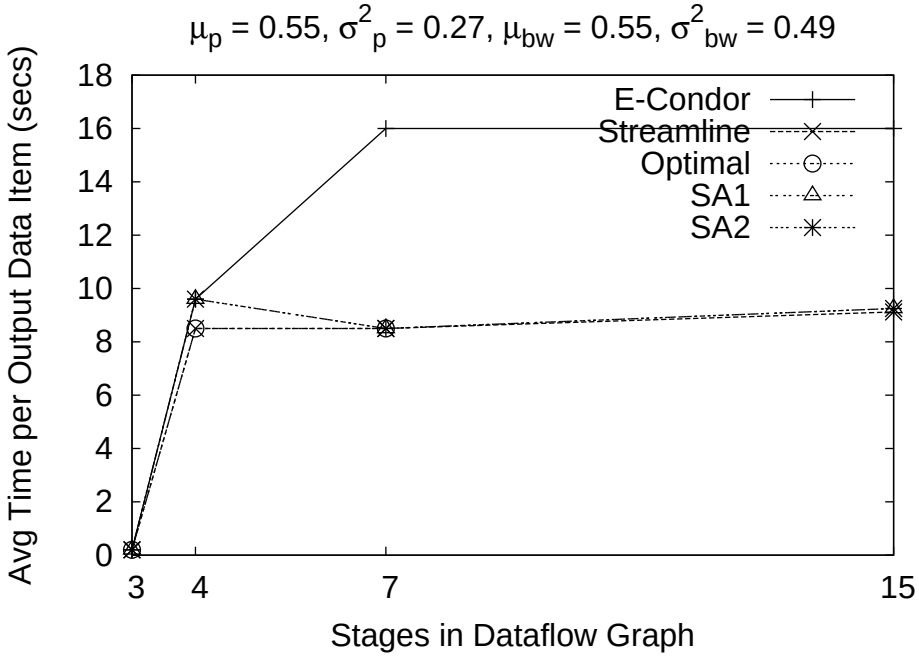
We build a representative dataflow graph for a video-based tracking application by combining our basic building blocks, *Collage*, *EdgeDetect*, *MotionDetect*, and *FD/FR* as shown in Figure 25. The application represents a scenario where streaming data from sensors are fed into an edge detector, merged near the source and are processed through successive stages of face detection, recognition and motion detection in order

to derive some higher level hypothesis.

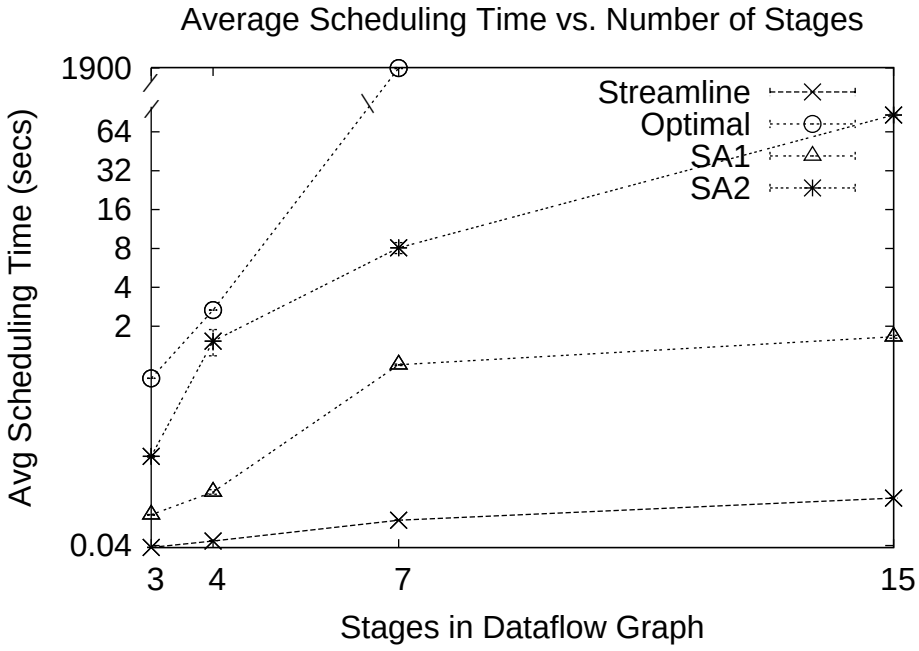
We measure the average time taken per output data item for a 3 stage (2 *EdgeD*, 1 *Collage*), 4 stage (2 *EdgeD*, 1 *Collage*, 1 *FD/FR*), 7 stage (4 *EdgeD*, 2 *Collage*, 1 *FD/FR*) and 15 stage (8 *EdgeD*, 4 *Collage*, 2 *FD/FR*, 1 *MotionD*) dataflow graph with different algorithms. We have performed the experiments on 15 nodes (1 processor in each node) with a particular choice of control parameters ($\mu_{bw} = 0.55$, $\sigma^2_{bw} = 0.49$, $\mu_p = 0.55$, $\sigma^2_p = 0.27$) that falls within the range used in the micro measurements, so as not to bias the experiment in favor of any one particular algorithm. Because of the computational complexity of the Optimal algorithm, we did not evaluate it for the 15 stage dataflow graph.

The results (see Figure 26(a)) show that (*i*) because of the introduction of a computation intensive stage *FD/FR*, all algorithms show an increase in the average time per output data item when the number of stages increases from 3 to 4; (*ii*) Streamline performs close to Optimal and about 12% better than SA1 and SA2 for 4 stage dataflow graph. This degradation in Simulated Annealing performance is attributed to our neighbor selection policy when the number of resources is more than the number of stages; (*iii*) For the 7 stage dataflow graph, Streamline outperforms E-Condor by a factor of 2; (*iv*) Streamline performs close (within 1%) to Simulated Annealing algorithms even when number of stages more than doubles from 7 to 15. This demonstrates that as the number of stages increase, Streamline is able to allocate resources better by taking into account non-uniform resource availability and the application needs. We also observe that SA1 performs as good as SA2 due to the computation intensive nature of the scalability graph, consistent with our findings from the micro measurements.

Finally, we compare the relative time taken by the different scheduling algorithms for the dataflow graph presented above. We measure the scheduling time of Streamline, SA1, and SA2 for the 3, 4, 7, and 15 stage dataflow graph. We also report the



(a) Effect of Increase in Number of Stages on Average Time per Output Data Item



(b) Effect of Increase in Number of Stages on Scheduling Time

Figure 26: Results of Scalability Tests

scheduling time for the Optimal algorithm for 3, 4, and 7 stage dataflow graphs. The numbers presented in Figure 26(b) are the average over 15 consecutive scheduling runs. From Figure 26(b), we observe that Streamline has a very small execution time (91 milliseconds for 15 stage dataflow graph). The execution time of SA1 (1.6 seconds) is more than Streamline by a factor of 18, whereas SA2 takes much longer (88 seconds) for the 15 stage dataflow graph. The variance in execution time across different runs was observed to be very small ($< 3.5\%$ for SA2, < 2.5 % for SA1, and $< 0.01\%$ in case of Streamline and Optimal). We conclude that Streamline has performance comparable to SA2 (within 1%) with much smaller execution time (by a factor of 1000), making it suitable for the dynamic environment of the grid.

4.5 Related Work

Scheduling in grid has primarily focused on providing support for batch-oriented jobs (See Nabrzyski, et al. [63] for a survey of current grid schedulers). A wide variety of meta-schedulers and resource brokers using Globus Toolkit [35, 34] have been developed by other research projects such as Condor[79], Legion[19], and Nimrod-G [16]. Most of these schedulers developed out of needs to support scientific batch-oriented applications. As we mentioned earlier, such schedulers do not address the needs of streaming applications. However, through the E-Condor architecture we have shown how these batch schedulers can be used to allocate resources for streaming applications. We have used E-Condor, in addition to Optimal and Simulated Annealing algorithms, as a baseline scheduler in experimental evaluation of Streamline.

Middleware efforts for streaming applications have started gaining attention in the grid community only recently. GATES [21] provides middleware-support for dynamically adapting a streaming application based on the observed processing rates in individual stages. A companion paper [20] describes a middleware for deploying the

stages of a generic application, processing stream data, to grid resources. The middleware uses the available communication bandwidth among the nodes to determine an assignment that will result in the best use of the resources under the assumption that earlier stages of the pipeline would need more communication bandwidth. Streamline is a more comprehensive framework for scheduling a streaming application taking into account the application characteristics as well as the computational resources available from the grid.

At some level, coarse-grain dataflow graphs of streaming applications resemble task-graphs that have been the focus of multiprocessor scheduling work from the 70's. The objective in task-graph scheduling is to minimize the total execution time of the application (represented as a task-graph) on a multiprocessor. The classical approach, called *list scheduling* [6, 24], (and its variants [39, 70, 37, 53, 82]), creates an ordered list of task-graph nodes by assigning them priorities based on certain properties. These priorities are then used to assign the tasks to processors such that each task is started at the earliest possible time commensurate with its priority. The specifics of the algorithms vary in the way priorities are assigned to task-graph nodes. The scheduling of streaming applications on the grid differs from task-graph schedulers in many ways. First, streaming applications are continuous in nature; therefore, all the stages of the application have to be scheduled to run concurrently. Second, the grid framework does not allow the level of control over the individual resources (for e.g., the operating system scheduler on a node) as assumed by such multiprocessor scheduling work. Third, there could be significant non-uniformity of computational resources (processing and communication bandwidths) leading to additional complexity in resource allocation.

Service composition for streaming applications has also received attention recently. SpiderNet [40] and a related paper [55] address QoS aware service composition problem for a dataflow graph in a Peer to Peer environment. SpiderNet [40] presents

a distributed framework for service composition and includes algorithms for service discovery, QoS aware service composition, and failure recovery. A related work [55] presents service composition algorithms for dataflow graphs with multiple sources and sinks. We address the problem of node selection for each stage of the dataflow graph whereas the research presented in these two papers address the issue of service selection for composition (assuming that individual services are already deployed). Therefore, Streamline can be used for initial service deployment and complement the above related works. We have also evaluated Streamline through an implementation in Globus Toolkit running on wide area network.

Stream processing has also been the focus of recent database research [23, 5, 13, 18]. Tools and techniques for the efficient handling of "continuous queries" on stream data are the objectives in such work, while our work focuses on scheduling streaming applications on grid resources.

4.6 Conclusion

In this chapter, I have presented a scheduling heuristic, *Streamline*[8, 9], that takes as input (a) computation and communication requirements of the various stages of a streaming application represented as a coarse-grain dataflow graph, (b) any application-specified constraints, and (c) the current resource (processing and bandwidth) availability. We have designed Streamline over an existing grid framework using Globus Toolkit [35].

I have compared Streamline with an Optimal placement and Simulated Annealing algorithms. In addition, to serve as a baseline for a comparative study, I also developed a streaming scheduling framework, called *E-Condor*, built using existing grid scheduler Condor[79]. We have performed experimental studies and shown that Streamline performs close (within 1%) to Optimal and outperforms E-Condor by nearly an order of magnitude on compute-bound kernels under non-uniform CPU

availability, and by a factor four on communication-bound kernels under non-uniform network bandwidth availability. Through two different choices of parameters for Simulated Annealing, I also show that the choice of neighbor selection and annealing schedule of a Simulated Annealing algorithm have a more profound impact for a communication intensive dataflow graph than for a computation intensive dataflow graph. The study has also shed light on the scalability of Streamline for large-scale streaming applications and shows that Streamline performs close (within 1%) to Simulated Annealing algorithm, with much smaller scheduling time (by a factor of 1000).

While this chapter presented Streamline scheduling algorithm for allocating high performance computing resources, the experiments were conducted using controlled environment to demonstrate the performance advantage of our scheduling heuristic. As mentioned previously, the resources required for a streaming application are typically distributed in a wide area environment. In the next chapter, I share my experience and experiment results from using Streamline in a wide area environment.

CHAPTER V

USING STREAMLINE IN A WIDE AREA ENVIRONMENT

In prior chapters, I presented a uniform programming abstraction (DFuse[52, 69]) and a runtime resource allocation mechanism (Streamline[8, 9]) for data streaming applications. Both these contributions make it easier to develop streaming applications in a heterogeneous and dynamic environment. In the last chapter, I presented the performance of Streamline in a controlled environment. However, it is clear that accesses to wide area environment resources are required for streaming applications. Wide area environment presents challenges because of the following reasons: (*i*) resources are shared by multiple users among multiple applications; (*ii*) The resources are not under any centralized control; (*iii*) The resource availability changes frequently.

In order to demonstrate the usefulness of Streamline scheduler in a wide area environment, I implement Streamline as a grid service in Globus Toolkit[35, 34] and evaluate the schedule generated by Streamline using Planetlab[68] resources. Planetlab provides a shared platform for distributed experiments, therefore, the resource availability is highly variable. Our intent in using Planetlab is twofold:

1. To demonstrate that Streamline can be implemented as a grid service facilitating the allocation of resources and launch of streaming applications from a high level specification.

2. To carry out a performance study of Streamline under conditions of dynamic resource availability and compare the performance to Simulated Annealing algorithm for compute and communication bound kernels. The performance study uses a large dataflow graph to understand the scalability of Streamline. Further,

unlike the controlled load conditions used in the previous chapter, the experiments on Planetlab use real resource availability information by contacting various other grid services configured on each machine.

I implement Streamline as a grid service on Planetlab[68] and conduct experiments using sixteen nodes of Planetlab. The study demonstrates that Streamline facilitates the allocation of resources and launch of streaming applications from a high level specification. My experimental results on Planetlab platform confirm that Streamline service performs close to Simulated Annealing (within 1%) under dynamic resource conditions.

I present the Streamline platform architecture in Section 5.1 and describe my implementation using grid services in Section 5.2. Experimental results are presented in Section 5.3 and I conclude in Section 5.4.

5.1 Architecture

Figure 27 shows the Streamline testbed and interaction of Streamline with other services. Streamline scheduling service runs on a single node of Planetlab and collects resource information from other nodes using other services. Once a dataflow graph is submitted to Streamline, Streamline contacts (*i*) Network Weather Service [88] for dynamic CPU load and end-to-end bandwidth information; and (*ii*) Ganglia [59] through Grid Information Service [26] (part of Globus Toolkit) for collecting static resource information (such as CPU speed). Once resource information is collected, Streamline scheduler performs the resource selection for each stage of the dataflow graph using the scheduling algorithm described in Chapter 4. After that, Streamline uses the job description files to submit each individual stage of the dataflow graph to a Grid Resource Allocation and Management (GRAM) [27] grid service running on the corresponding machine. GRAM service uses the job description file to launch the job on the allocated host. Once all the stages are launched on the allocated machine,

the application stages use a registry service to discover each other and establish connections according to the dataflow graph specification.

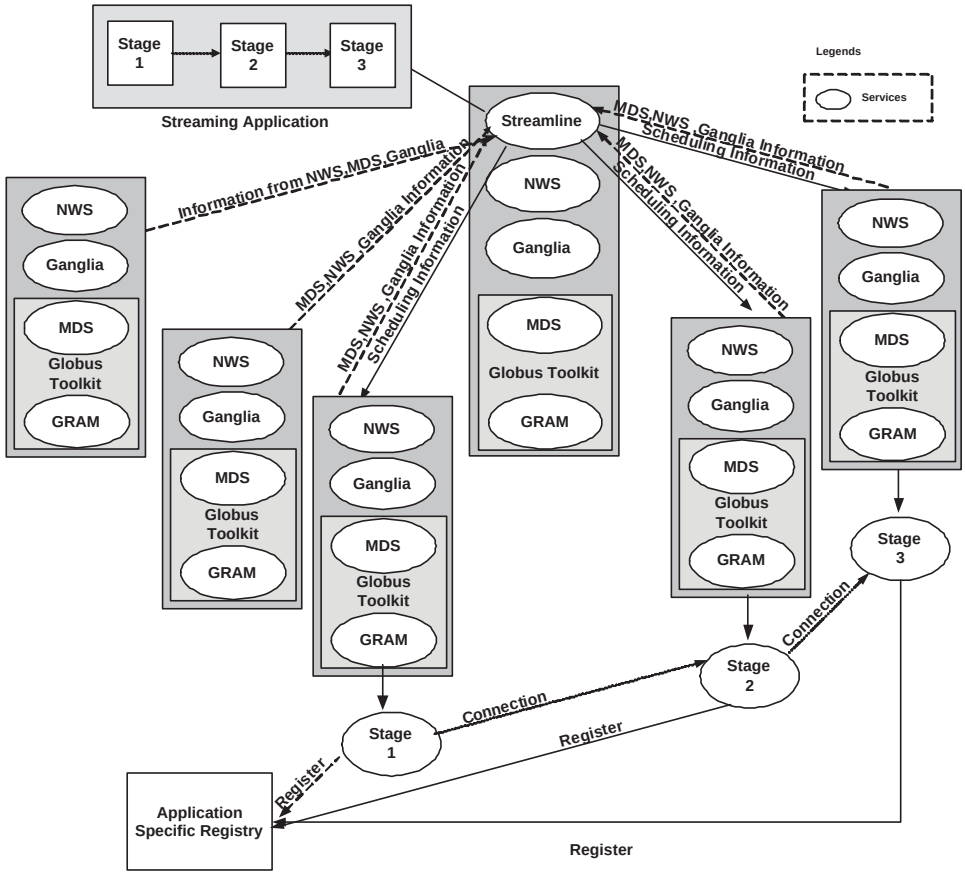


Figure 27: Streamline Testbed

Please note that after Streamline launches the application stages on the allocated machines, the stages are free to use any communication protocol. The application specific registry is used to discover the locations of the stages, and after that the stages communicate using an application specific protocol.

5.2 Implementation

We select sixteen Planetlab nodes (as shown in Figure 28) as our experimental platform. As part of our implementation, we installed and configured all the necessary software (such as Globus Tookit[35], Ganglia[59], Network Weather Service[88]) on the sixteen nodes of Planetlab. For collecting static information about the resources,

a SAX parser was used to parse through the query results provided by Grid Information Service and communicate the necessary information (such as hostname, available memory, OS, IP address) to the Streamline service. Network Weather Service (NWS)[88] was used to get real-time information about the available CPU and end-to-end bandwidth between the machines. We wrapped C implementation of the NWS APIs using Java Native Interface (JNI) and deployed it as a grid service.

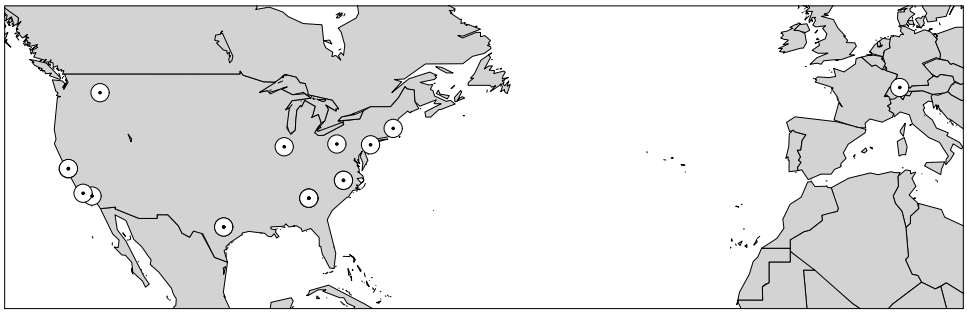


Figure 28: Planetlab Nodes Used for Experiments

We created grid services for Streamline and Simulated Annealing (SA2, described in Section 4.4) scheduling algorithms. As we observed in Section 4.4, Simulated Annealing performs close to Optimal and is used as a benchmark for comparison with Streamline. Both Streamline and Simulated Annealing services interact with other services for gathering resource availability information and for launching jobs after scheduling. The inputs to the schedulers are (*i*) a job description file and (*ii*) a dataflow graph specification file. The job description file is passed by Streamline to GRAM[27] to launch jobs, and the dataflow specification file is used by Streamline for scheduling. The computation and communication requirements of the various stages are fixed at the start of each experiment, as determined by application profiling and shown in Table 4 in Chapter 4

5.3 Experimental Setup

In order to model the CPU usage for each stage of the dataflow graph, we use x86 assembly language instructions that consume CPU cycles commensurate with the numbers reported in Table 4 for each of the functions on a lightly loaded machine. The actual computation time is dependent on the load on the machine. Similarly, the stages communicated data according to the communication requirements in Table 4. In contrast to our experiments in Section 4.4, we do not model resource availability since the Planetlab shared environment has dynamism in resource availability. Therefore, we use actual computation and communication commensurate with the numbers in Table 4 for our experiments.

In order for the stages to be executed on the allocated machines after scheduling, the application code and input data need to be transferred to the executing machine. In our testbed of sixteen machines, we pre-loaded all the hosts with the executable code, and passed the locations of the executable in a job description file to Streamline. Since our goal was to evaluate the scheduling heuristic, we adopted this simple solution for our testbed. In a production environment, grid services can be used for file staging before a job is launched on the allocated machine.

5.3.1 Micro Measurements

We use the same compute bound (Figure 17) and communication bound (Figure 21) kernels for our micro measurements on Planetlab. We selected six nodes of Planetlab for our micro measurements. A separate Planetlab machine is used for running the scheduling services. For each kernel, we use Streamline and Simulated Annealing as the scheduling algorithms for comparison. The metric for comparison is the average time taken per output data item averaged over 100 data items. We run the experiments once every thirty minute for a four hour period in order to observe the effect of any changes in the environment. We use the time when we run the experiment

as the control variable on the x-axis. Therefore, we have eight data points for each experiment. Based on the observed resource availability, we also compute the mean and variance in processing and bandwidth availability for each experimental run.

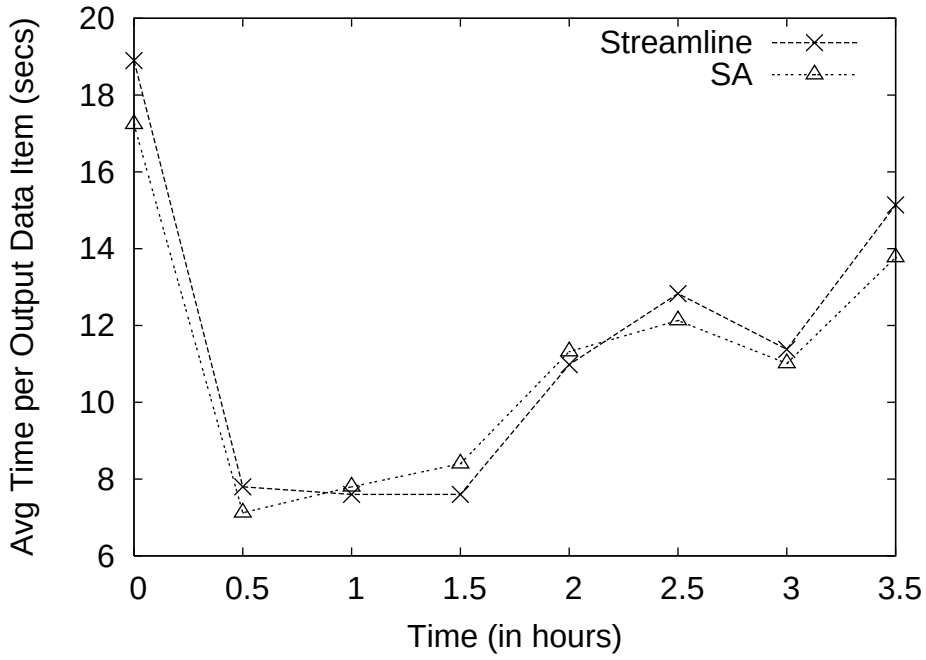


Figure 29: Micro Measurements using Compute Bound Kernel on Planetlab

As can be seen from Figure 29, Streamline performs close to Simulated Annealing for compute bound kernel.

For communication bound kernel (Figure 30), we observe that Streamline even outperforms our Simulated Annealing algorithm for some instances. We believe that the poor performance of Simulated Annealing is due to the variability in resource conditions between the separate runs.

In Figure 31, we show the observed mean and variances in CPU and bandwidth availability across the six nodes used for this experiment. As can be seen from the figure, the mean and variance of CPU availability remain stable throughout the four hours we run these experiments. Although the mean in bandwidth availability remains stable, we observe that the variance in bandwidth availability is unstable. We believe this variance in bandwidth availability to be the reason for the relatively poor

performance of Simulated Annealing for communication bound kernel. The transient load conditions on Platetlab resources lead to one instance where Streamline performs worse compared to other times in Figure 30.

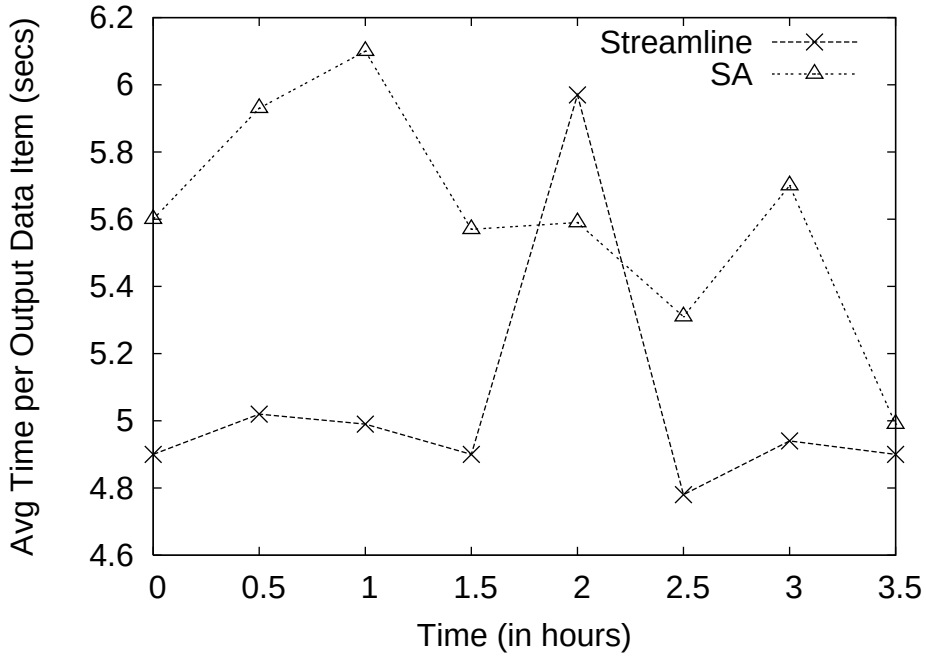


Figure 30: Micro Measurements using Communication Bound Kernel on Planetlab

5.3.2 Scalability

We use the 15 stage dataflow graph (Figure 25) for our scalability experiment. We measure the average time taken per output data item for a 3 stage (2 *EdgeD*, 1 *Collage*), 4 stage (2 *EdgeD*, 1 *Collage*, 1 *FD/FR*), 7 stage (4 *EdgeD*, 2 *Collage*, 1 *FD/FR*) and 15 stage (8 *EdgeD*, 4 *Collage*, 2 *FD/FR*, 1 *MotionD*) dataflow graph with Streamline algorithm. The experiments were performed on 15 Planetlab nodes. We use an additional Planetlab node for running the scheduling services.

From Figure 32, we observe that the effect of increasing the number of stages on Streamline schedule is similar to our earlier experiment in Figure 26(a). The average time per output data item increases when the number of stages increases from 3 to 4 due to the introduction of a compute intensive FD/FR stage. Streamline performance

slowly degrades as the number of stages is increased to 15.

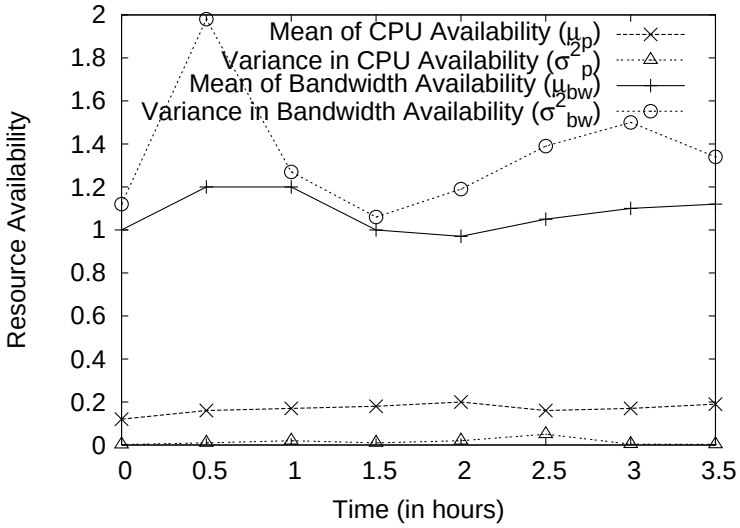


Figure 31: Mean and Variance of Observed Resource Availability in Planetlab

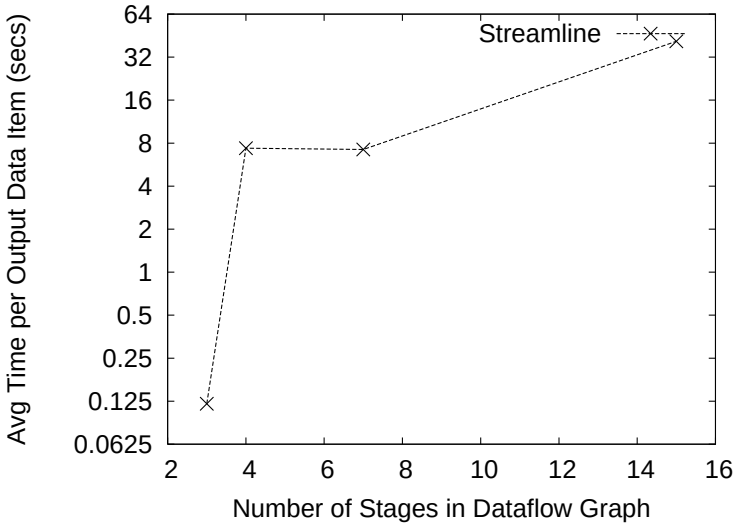


Figure 32: Scalability of Streamline on Planetlab

For comparison, we also run Simulated Annealing algorithm for the 15 stage dataflow graph and observe that while Streamline schedule takes 41.31 seconds, Simulated Annealing schedule takes 42 seconds for average time per output data item. This validates that Streamline performs close to the computationally intensive Simulated Annealing algorithm for large dataflow graph in a wide area environment. Since

we use Simulated Annealing for only the 15 stage dataflow graph for validating the scalability of Streamline, the single data point for Simulated Annealing is not shown in Figure 32.

5.4 Conclusion

In order to demonstrate the usefulness of Streamline scheduler in a wide area environment, I implement Streamline as a grid service in Globus Toolkit and automate the execution of streaming application dataflow graph from a high level specification. I evaluate the schedule generated by Streamline on Planetlab[68] resources and confirm that Streamline performs close to Simulated Annealing algorithm under dynamic resource availability.

While Streamline does the placement for such streaming applications well (in both controlled and wide area environment), it is clear that the application dynamics may result in the computation and communication characteristics of the application changing over time. Perhaps even the dataflow graph of the application could change over time with the addition and deletion of new stages to the pipeline. If the application characteristics that are profiled and used in the placement are considered the "typical" (for e.g., mean) values for the respective stages, then the mapping given by Streamline would result in an acceptable level of performance despite this dynamism. Nonetheless, It is important to consider the impact of the application dynamism and the consequent adaptation of the scheduling heuristic. In the next chapter, I describe my solutions for dynamic resource management.

CHAPTER VI

DYNAMIC RESOURCE MANAGEMENT

6.1 Motivation

In prior chapters, I described Streamline scheduling heuristic [8, 9] that makes it easier to allocate high performance computing resources for a data streaming application. However, application and resource dynamism may require re-evaluation of the mapping generated by Streamline in order to deliver required QoS to the application. The dynamism arises primarily for the following reasons: (*i*) resource availability in the Grid changes with resources being added or removed from the target environment; and (*ii*) the initial estimate of application requirement may change as a result of the computation being performed. For instance, in the distributed surveillance application, the computation and communication requirements of a particular node may change based on the amount of activity in the area being monitored. Since it is critical to identify events as soon as they happen, the resource management system needs to adapt the scheduling decisions based on the dynamism.

In this chapter, I address the dynamism that comes from changes in application requirements or resource availability. The application data-flow graph may also change with addition and deletion of new stages to the graph; however, addressing such changes to the dataflow graph are outside the scope of this dissertation.

In chapter 3, I have summarized how DFuse addresses the dynamism in sensor network environment via fusion point migration and algorithms for role assignment that take power consumption and data transmission cost into account. There are architectural similarities between dynamic scheduling described in this chapter and DFuse. For example, both start with an initial placement algorithm followed by

subsequent adaptation based on observed dynamism in the environment (based on cost functions). However, the algorithms in this chapter focus on specific requirements in allocating high performance computing resources. I use Streamline as the initial placement algorithm. Also the cost function used in this chapter takes into account computation and communication requirements of the application, whereas, in DFuse, communication and power consumption played key roles in migration decisions. In addition, I use a feedback loop in the dynamic scheduling algorithm to adapt the result of the scheduling based on application feedback. This work builds on top of the Streamline work described in previous chapters.

6.2 Problem Statement

I consider the same system model as described in Section 4.1 but add dynamism to application requirements. The estimate of the execution cost (w_i) of stage s_i and the estimate of data transmission cost ($c_{i,j}$) for edge (i,j) may change with time. In addition, the target computing environment may change based on usage of shared resources. I use the information service [26] and the Network Weather Service (NWS) [88] to discover current resource availability.

The *objective function* of the dynamic scheduling problem is to provide the best possible throughput (the rate at which data items are produced by the output stage) throughout the lifetime of the application. The dynamic scheduler should also minimize the overhead of making the scheduling decision.

6.3 Requirements

In this section, I describe some of the system requirements for dynamic resource management.

6.3.1 High Availability

In the design space of possible dynamic scheduling algorithms, at one extreme, there are algorithms that would require stopping the application periodically for a certain amount of time. During this time period, scheduling decision can be made again based on the new data available, and the application dataflow graph is instantiated again. At the other end of the spectrum, there are solutions that do not require stopping the application at any point. If it takes very small amount of time to stop the application, make the scheduling decision, and restart the application (perhaps for small dataflow graphs), the first extreme may be perfectly reasonable. However, because of the real-time requirements, such a solution would be infeasible even for a reasonably sized dataflow graph. Ensuring high availability is a key requirement for this application class because of the time critical nature of the decisions being made.

6.3.2 Preserving Application State

Vision processing applications typically build various models (such as a model of the background) based on observed data in order to identify interesting events. Each stage of the application has an associated state. Any dynamic adjustment to the scheduling decision needs to preserve the application state even when an application stage is migrated to a new node.

6.3.3 Correctness

The correctness requirement of these applications can be formulated as: "interesting events should be identified as soon as they happen". For example, an application may skip processing certain frames based on its logic and results from prior computation; however, the system should not drop frames while it is making dynamic scheduling decisions in the absence of any special hints from the application. Dropping data may lead to missing interesting events and is unacceptable.

6.3.4 Common Abstractions

Developing streaming applications is difficult because of all the challenges involved in distributed programming. Providing a common abstractions that would allow automated dynamic scheduling decisions for these applications would make it easier for domain experts to build them.

6.3.5 Handling Failures

Various kind of failures (such as node failures, network failure, application QoS failure) are possible during the lifetime of an application. Any system should be designed keeping in mind possible failure scenarios.

6.4 System Architecture

A solution to the dynamic scheduling problem consists of (*i*) monitoring, (*ii*) algorithm for making re-scheduling decisions, and (*iii*) adjusting the application dataflow graph based on the results from the scheduling algorithm.

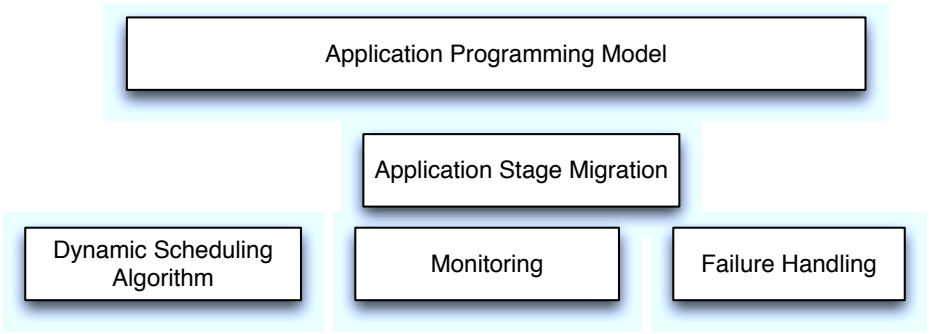


Figure 33: Dynamic Scheduling System Architecture

6.4.1 Monitoring

Monitoring is an inherent component of a dynamic scheduler. Monitoring can be done at different granularity with or without application support. Monitoring infrastructure determines when resources are under-utilized or over-utilized. It also determines

when the application can make use of additional resources.

Since one of our goal is to use existing grid functionality, I use the grid information service [26] to query static information (machine architecture, CPU speed, amount of memory, and hardware configuration) about available resources. I use Network Weather Service (NWS) [88] to obtain dynamic information(such as estimate of available processing cycles and end-to-end network bandwidth) about the resources.

I assume that current computation and communication requirements for each stage of the application are made available. This information may either be provided by each stage of the application in the form of *resource properties*, or it can be inferred automatically based on the input/output buffer sizes and the processing rates of the different stages of the application dataflow graph.

6.4.2 Making Rescheduling Decision

Given that the application requirements and resource availability are available continuously, one of the challenges is in deciding when the scheduling decision is re-evaluated. Following are some of the design choices:

(*i*) Periodic Evaluation: The scheduling decision can be evaluated periodically with statically or dynamically determined period. Having a very small re-evaluation period will allow the application to adapt quickly to the dynamic environment; however, it would lead to higher overhead (algorithm overhead and stage migration overhead) in the presence of limited dynamism. On the other hand, having a high re-evaluation period has less overhead, but may not adapt to the dynamism of the environment. The re-evaluation period can itself be dynamically adjusted based on observed changes in the environment.

(*ii*) Application QoS Triggered Scheduling: The application can trigger a new scheduling decision based on the degradation in the quality of service. As another

alternative, the system infrastructure may monitor the application QoS and trigger a scheduling decision whenever the application QoS falls below a threshold.

(*iii*) Central Decision Vs. Distributed Decision: In one approach, a central policy can decide when to re-evaluate the scheduling decision. In another possible approach, each node of the dataflow graph could independently decide when to re-evaluate the scheduling decision based on locally observed application QoS. A distributed decision model is highly scalable as the dataflow graph size increases.

Our system architecture uses a distributed decision model where each node of the dataflow graph decides how frequently the scheduling decision should be made. Each node makes this decision based on locally observed data and a cost function. I compare this approach with a centralized decision model where Streamline is used periodically to make scheduling decisions.

6.4.3 Programming Abstraction to Facilitate Dynamic Scheduling

Our system model for dynamic scheduling assumes that each application stage is represented as a service. The individual service will need to be restarted on the newly allocated node after migration. Streaming applications also require establishing appropriate input/output connections once migration is complete. There may be data in-transit between the stages; the migration infrastructure needs to ensure that none of the data in-transit is lost.

I address the above issues as follows: (*i*) In case the application service is not running on the new node, the service code is downloaded and instantiated. The migration algorithm takes into account the cost of service instantiation in making its migration decision; (*ii*) I use application provided *entry-points* in order to start the migrated services on the new nodes and establish input/output connections. In order for the system to make dynamic scheduling decision, I assume that an application implements certain well-designed entry-points that can be used by the resource

management system. In particular, our system architecture requires the following entry-points to be provided by an application:

initialize(state): Every stage of the dataflow graph implements an initialize function that takes in serialized state information about the stage. This state information can be used by the application to reconstruct its state once the stage is migrated to a new node.

captureState(): The scheduling infrastructure can call the captureState function in order to capture the state of the application stage. This state is transferred to the new node as a result of the scheduling algorithm evaluation.

start(inputs, outputs): This function is used to start computation on the dataflow graph. Each stage consists of gathering data from the input, processing the data, and sending the data to the output stage. The scheduling infrastructure provides the input and output connections to a stage as specified by the application dataflow graph.

stop(): This function is called by the resource management system to stop the computations being performed by a particular application stage. Once this entry-point is called, the application stops gathering data from the input stage and only completes processing the current data. Once the current data processing is finished (or a timeout is reached), the system calls captureState to capture the application stage data, migrate the stage to the new node and restarts the computation after establishing the input/output pipeline.

The dynamic scheduler uses these application *entry-points* in order to initialize and restart the application services on the new nodes. In order to ensure that no data stream is lost during migration, I use a hand-off protocol. The original node will continue running the service until migration is complete, and will forward any unprocessed data to the new service node after migration. This protocol is necessary in order to ensure that service migration happens seamlessly.

When an application stage is being migrated to a new node, no "useful" work is being performed. Various optimizations are possible to overlap "migration time" with the computation being performed on the old node. For example, the application state can be migrated in parallel, and the stage can be instantiated on the new node in parallel. Before starting the computation on the new node, any possible additional state could also be sent. Evaluating the trade-offs in these optimization policies is part of future work.

6.4.4 Scheduling Algorithm

I evaluate the following scheduling algorithms: (*i*) Centralized Streamline algorithm that re-evaluates the scheduling decision periodically, and (*ii*) a distributed algorithm where each node dynamically decides on the best possible placement of its computation based on local information and a cost function. I use the static placement generated by Streamline as the baseline and compare the overhead as well as observed QoS for these two scheduling algorithms using controlled experiments.

6.4.4.1 Periodic Streamline Algorithm

As shown in chapter 4, Streamline algorithm does a good job with the initial placement of the streaming application dataflow graph. However, the initial placement of streamline may become sub-optimal as resource availability changes. Therefore, I use the periodic Streamline algorithm to address the dynamism in the environment. In this approach, Streamline algorithm is run periodically at a central location. The algorithm starts with a fixed periodic evaluation, and adapts the period based on the observed QoS of the application.

Based on the results from the re-evaluation of the Streamline algorithm, stages are migrated to new nodes. The application dataflow graph is stopped for a brief period while the migration is in process. Both the computation and data associated with each stage are packaged and migrated to the new node.

Periodic Streamline is a centralized algorithm and does not satisfy the high availability goal mentioned earlier. However, it provides a good baseline for dynamic scheduling algorithm performance. Moreover, when the dataflow graph is small, the overhead of this algorithm may be acceptable. Through experiments, I compare the performance of this algorithm with the one time placement generated using Streamline.

6.4.4.2 Distributed Dynamic Scheduling Algorithm

Periodic streamline algorithm is centralized and can have high overhead. In order to address the limitations of the periodic Streamline algorithm, I propose a distributed scheduling algorithm. Each node of the dataflow graph computes the cost and benefit of migrating each of the computation on its node to a neighbor based on a cost function. Expected benefit of migrating the computation to a neighboring node is determined by calculating estimated throughput on the new node and the throughput on the current node. The throughput on a node is estimated by calculating the time it will take to process a single frame (similar to the Streamline algorithm).

The cost ($A\ (n_j,\ s_i)$) of allocating a resource node $n_j \in R$ to a stage s_i is computed by summing up the estimated computation and communication cost (in time units) for this particular assignment and is represented as

$$A(n_j, s_i) = ecycles_i/proc_j + \sum_{s_k \in pred(s_i)} ecomm_{k,i}/b_{k,i} + \sum_{s_k \in succ(s_i)} ecomm_{i,k}/b_{i,k}$$

where $ecycle_i$ is an estimate of the average amount of CPU cycles required by stage s_i; $proc_j$ is an estimate of the available CPU cycles on node n_j per second; $ecomm_{i,k}$ is the average amount of data transfer (in bytes) from stage s_i to stage s_k; and $b_{i,j}$ is an estimate of available end-to-end network bandwidth (in bytes per second) between node n_i and node n_j. The cost function A captures the time it takes by a particular stage to process a single data item on a particular node by summing up the computation and communication time.

The benefit of migration is calculated as the difference in the cost function on the new node Vs. the current node. Each node of the dataflow graph periodically gathers resource availability information from its neighboring nodes, and calculates the benefit of migration to one of the neighbors. If the benefit exceeds a predetermined threshold, the node initiates an "intention to migrate the computation" to the new node by sending a message to the new node. The new node may accept or deny the request based on all the requests it receives. Upon receiving confirmation from the new node within a specific timeout period, the migration process is started using a hand-off protocol. Once the migration is complete, the node instantiates the computation on the new node using the application provided entry-points mentioned earlier.

Each node of the dataflow graph computes the cost of migrating its computations on various neighbors. Based on the cost function, a given node decides the best node and migrates the stage to the best node. Because of the possible time delay between migrating a stage and the decision made for migration, it is possible that more than one stage decides to migrate the computation to the same node thereby degrading overall throughput of the application. In order to prevent this, the new node makes the final decision about which stage it accepts based on the expected gain in overall application quality of service. If admitting the new stage of the dataflow graph exceeds the resources of the node, the migration is denied and the original node sends its intention to migrate to the second best node, and so on.

The overhead of migration is calculated as the sum of the cost of migration, the cost of scheduling, the cost of initializing the computation on a new node, and the cost of connection establishment. Since streaming applications are long running applications, the migration overhead can be amortized over the expected gain, provided that the estimated bandwidth and CPU availability on the new node is sustainable over a relatively long period. Therefore, I take the estimate of available network and bandwidth availability on a new node over a minimum time period. Also in order to

avoid thrashing, I define a minimum time period for which a stage must be running on the current node before it can be rescheduled. This period is determined by the migrating node based on the scheduling overhead. In addition, I define a minimum quality of service threshold below which an application stage may force migration of the computation to a new node in order to avoid being stuck in a local minima for too long.

6.4.4.3 Feedback to Scheduling Algorithm

Various configuration parameters affect the performance of the distributed dynamic scheduling algorithm. Some of the configuration parameters are: (*i*) the periodicity with which each node re-evaluates the scheduling decision; (*ii*) the minimum expected gain in application quality of service that triggers stage migration; and (*iii*) the minimum duration for which a stage needs to run on a particular node before it may re-evaluate the scheduling decision. In our system architecture, each of these parameters has a fixed value in the beginning. However, each of these parameters is adapted dynamically based on application feedback and observed performance of a node. Each stage of the dataflow graph dynamically adapts these parameters based on local information.

6.4.5 Failure Handling

Failures can be handled either by a central policy or a distributed policy. From the point of view of scalability, I discuss a few distributed policies to handle various types of failures.

(*i*) Application stage migration failure: Failure during migration of an application stage is handled by the system architecture. The original node monitors the new node until the application processing starts. If the application stage is not started on the new node within a specified time period, the original node tears down the computation on the new node and takes charge of the stage again.

(*ii*) Node and Network failures: Node failures can be handled by our system by active monitoring of node status and a distributed co-ordination protocol among different nodes of the dataflow graph. Our work focuses on evaluating the scheduling algorithm trade-offs. Adapting the scheduling decision based on node and network failures is part of future work.

(*iv*) application QoS failures: Our scheduling algorithm is triggered based on application QoS failures. For example, whenever application throughput goes below a threshold, the scheduling decision is re-evaluated. Application QoS failures can be either determined by the system automatically or can be specified by the application to the resource management system.

6.4.6 Discussion

I have presented three different scheduling algorithms so far: (*i*) static Streamline, (*ii*) periodic Streamline, and (*iii*) the distributed scheduling algorithm. The static Streamline algorithm does one time placement of streaming application dataflow graph and is expected to perform well only when the application requirements and resource availability are static. The periodic Streamline is expected to provide better performance by re-evaluating the scheduling decision periodically. However, the periodic Streamline algorithm may lead to high overhead because the complete dataflow graph may be re-instantiated as a result of scheduling decision. The distributed scheduling algorithm is expected to be more scalable as the size of a dataflow graph increases.

Since the distributed scheduling algorithm makes local decisions (unlike the periodic Streamline algorithm), the resulting schedule may get stuck in local minima. To address this issue, each node of the dataflow graph forces migration to a new node (even if the expected cost of running the computation on new node is higher) if the application quality of service goes below a configurable threshold. In addition, in

order to avoid oscillations, a migrated stage must run for a minimum amount of time (configurable) on the new node before it is considered for rescheduling. The configuration parameters of the algorithm can themselves be dynamically adapted. Evaluating the impacts of these configuration parameters is an avenue for future work.

6.5 Evaluation

Periodic Streamline I implement the periodic streamline algorithm as a grid service. One central host runs this algorithm periodically and calculates the incremental benefit of the new schedule compared to the current schedule. I use the grid information service [26] to query static information (machine architecture, CPU speed, amount of memory, and hardware configuration) about available resources and Network Weather Service (NWS) [88] to obtain dynamic information(such as estimate of available processing cycles and end-to-end network bandwidth) about the resources. Also the algorithm takes as input, the current application requirements for each stage of the dataflow graph. Based on these inputs, the Streamline algorithm is run and the incremental benefit (improvement in application throughput) is calculated. If the benefit exceeds a pre-determined threshold, application stages are migrated to the new nodes and the connections are re-established.

In order to simplify the scheduling period decision, I assume that each node of the dataflow graph makes a periodic scheduling decision or re-evaluates the decision based on feedback from the application. I do not evaluate dynamic adjustment to the scheduling re-evaluation period as part of this work.

Distributed Scheduling Algorithm The distributed scheduling algorithm is implemented as a separate service that runs on each node of the dataflow graph. This service takes current resource availability of the neighboring nodes, and the current application requirements as input and takes care of instantiating application

stages on appropriate nodes, monitoring the performance of the application, making distributed scheduling decisions, and migrating the stage to a selected new node. My current implementation is limited to evaluating the performance of the dynamic scheduling algorithm. Therefore, it focuses on re-evaluating scheduling decision periodically and evaluating the application performance once stages are migrated to the new nodes. Evaluating the performance of stage migration (using application provided entry-points) is an area for future research.

For evaluating the performance of the dynamic scheduling algorithms, I borrow ideas from the Streamline algorithm evaluation in section 4.4. I use the compute and communication bound kernel under various resource configurations to quantify the benefits of periodic Streamline and distributed scheduling algorithm over static Streamline.

6.5.1 Micro Measurements

I use the same compute and communication bound kernels used in Section 4.4.6 for micro measurements. The scheduling algorithms (static Streamline, periodic Streamline, and distributed algorithm) map the 4 stages of the pipeline to the 4 nodes of the cluster. As mentioned earlier, I assign the settings (CPU load and network contention) on the processor and the links commensurate with the control variables for a particular resource configuration. The metric for comparison of different scheduling algorithms is the average time taken per output data item averaged over 100 data items. The distributed algorithm has a minimum threshold of 5% performance improvement before migrating a particular stage.

6.5.1.1 Compute-Bound Kernel

Figure 34 shows the compute-bound kernel. The dataflow graph consists of the following functions: (*i*) **Collage**: A simple concatenation of two images to produce a composite output; (*ii*) **EdgeDetect**: An algorithm to determine the boundaries of

objects in an image; (*iii*) **MotionDetect**: An algorithm that computes the magnitude and centroid of inter-frame differences in images to derive inferences on motion; and (*iv*) **FD/FR**: A compute-intensive algorithm to detect and recognize faces in an image that is based on skin tone analysis. I compare the performance of the 3 algorithms as resource availability is changed between various configurations picked from Section 4.4.

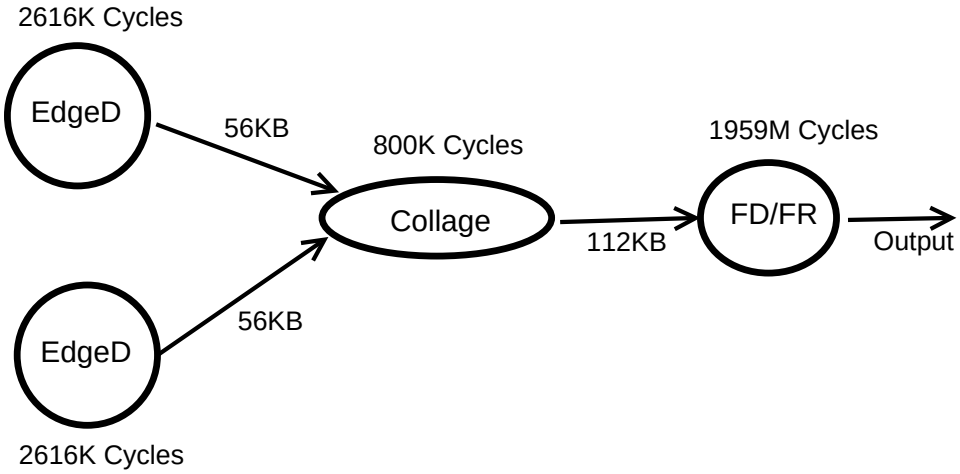


Figure 34: Compute-bound Kernel

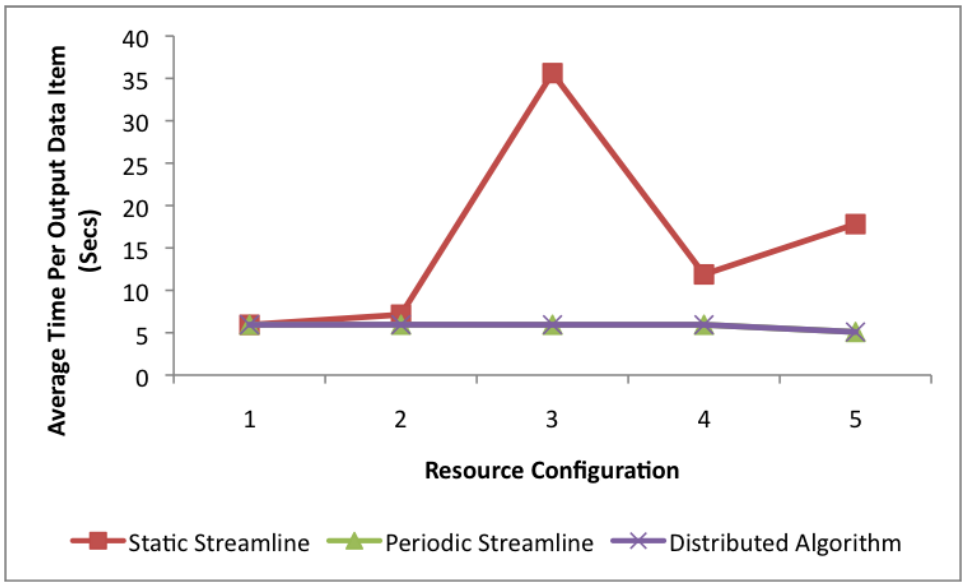


Figure 35: Performance of Different Scheduling Algorithms on Compute-Bound Kernel

As shown in Figure 35, the performance of static Streamline algorithm varies greatly as resource configuration is changed. On X-axis, I show five different resource configurations. The resource configurations vary in their CPU and bandwidth availability and are picked at random from the various resource configurations I experimented with in Section 4.4. On the Y-axis, I plot the throughput of the application measured by the execution time (in seconds) for producing a single data item. As can be seen, periodic Streamline is able to deliver a consistent application performance by re-evaluating the scheduling decision periodically and allocating best possible resources for each configuration. In one configuration, periodic Streamline is able to perform six times better than static Streamline. The performance of distributed algorithm is indistinguishable from the periodic Streamline algorithm because the dataflow graph is small.

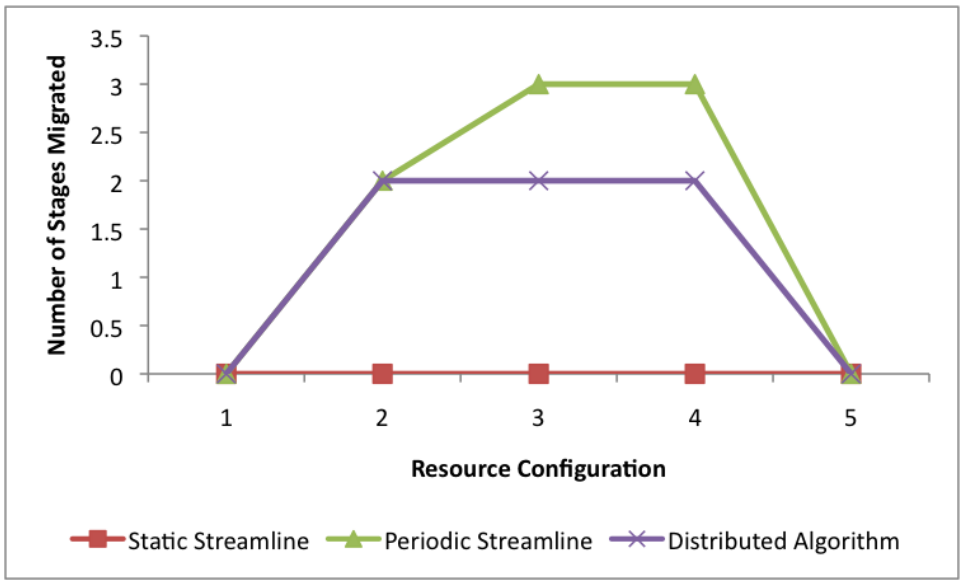


Figure 36: Number of Stages Migrated by Different Scheduling Algorithms on Compute-Bound Kernel

In Figure 36, I show the overhead of the different algorithms as measured by the number of stages that are migrated as resource configurations change (while application requirements remain the same). As resource configuration is changed, each

algorithm requires different number of stages to be migrated. The static Streamline algorithm, by definition, has zero overhead because it does not re-evaluate the scheduling decision. On the other hand, periodic Streamline may require the complete dataflow graph to be re-instantiated as resource configuration changes. As shown in Figure 36, periodic Streamline algorithm has 50% more overhead (requiring 3 stages to be migrated) than the distributed algorithm (requiring 2 stages to be migrated) as resource configurations change from configuration 2 to 3 and from configuration 3 to 4. This demonstrates that the distributed algorithm is able to achieve similar performance as periodic Streamline but with much less overhead for compute-bound kernel.

6.5.1.2 Communication-Bound Kernel

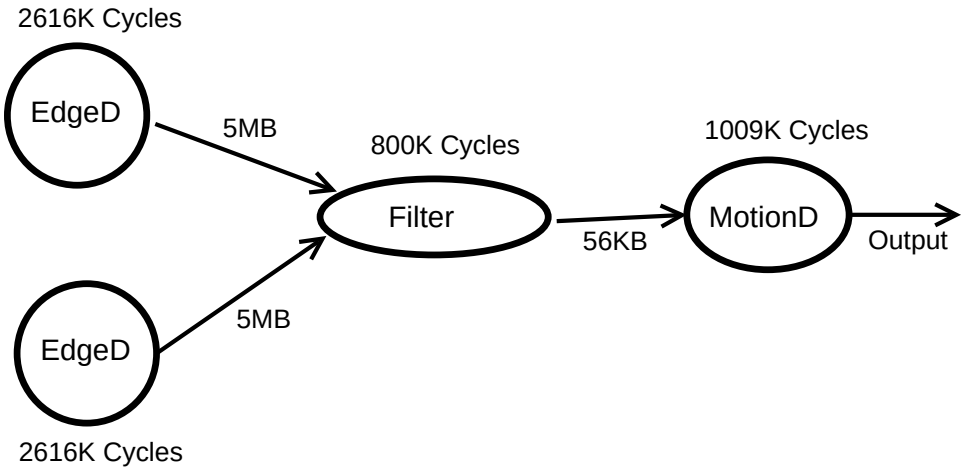


Figure 37: Communication-bound Kernel

Figure 37 shows the communication-bound kernel used for this set of micro measurements. In this kernel, a synthetic *Filter* function is used. The filter function receives large amounts of data (video images plus their boundaries) from two *edge detectors*. The filter function selects the subset of the input to send on to a *motion detector* for higher level inference.

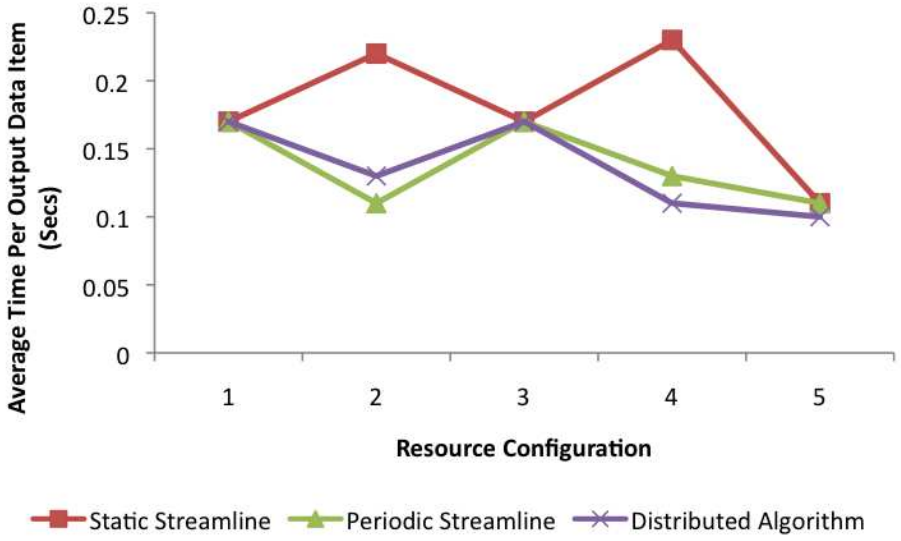


Figure 38: Performance of Different Scheduling Algorithms on Communication-Bound Kernel

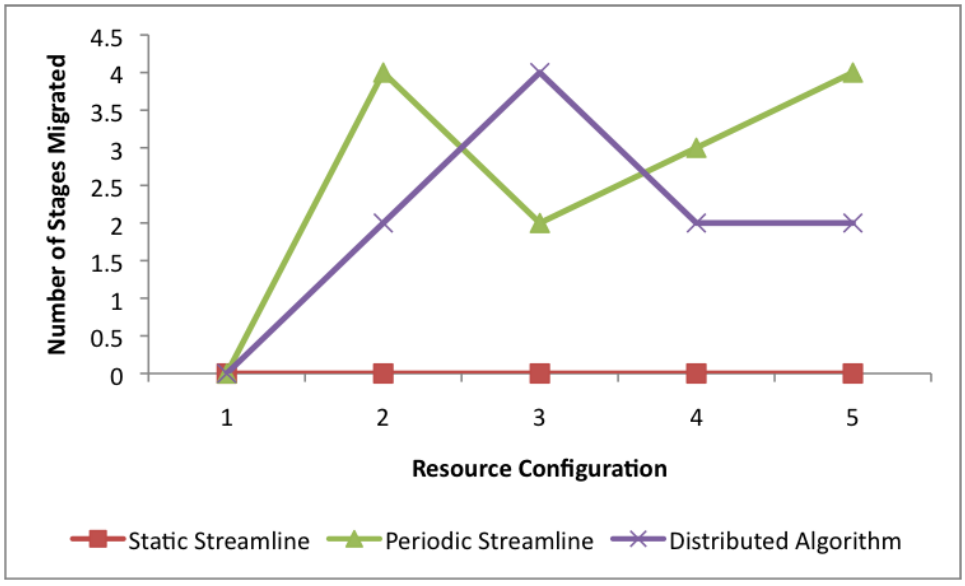


Figure 39: Number of Stages Migrated by Different Scheduling Algorithms on Compute-Bound Kernel

Figure 38 shows the performance of the different algorithms for the communication-bound kernel. The results confirm that the performance of static Streamline algorithm varies greatly as resource configuration is changed. But periodic Streamline is able to deliver a consistent application performance by re-evaluating the scheduling decision periodically and allocating best possible resources for each configuration. Moreover, performance of the distributed algorithm is very close to the performance of periodic Streamline algorithm.

Figure 39 shows the overhead of the three algorithms for communication-bound kernel. On an average across the five resource configurations evaluated, the distributed algorithm results in less number of stages to be migrated compared to the periodic Streamline algorithm. However, because of the small dataflow graph, in one instance distributed algorithm results in migration of more stages compared to periodic Streamline algorithm.

Through these micro measurements, I establish that the distributed algorithm performs close to periodic Streamline algorithm with less overhead and gives six times more throughout when compared to static Streamline for the experimental configurations. It is clear that static Streamline is not sufficient under varying resource conditions. I therefore evaluate the trade-offs between the periodic Streamline and the distributed algorithm further in the scalability study below.

6.5.2 Scalability

For the scalability study, I consider the same video-based tracking application used in Section 4.4.7. I build a representative dataflow graph for video-based tracking application by combining our basic building blocks, *Collage*, *EdgeDetect*, *MotionDetect*, and *FD/FR* as shown in Figure 40. The application represents a scenario where streaming data from sensors are fed into an edge detector, merged near the source and are processed through the successive stages of face detection, recognition and

motion detection in order to derive some higher level hypothesis.

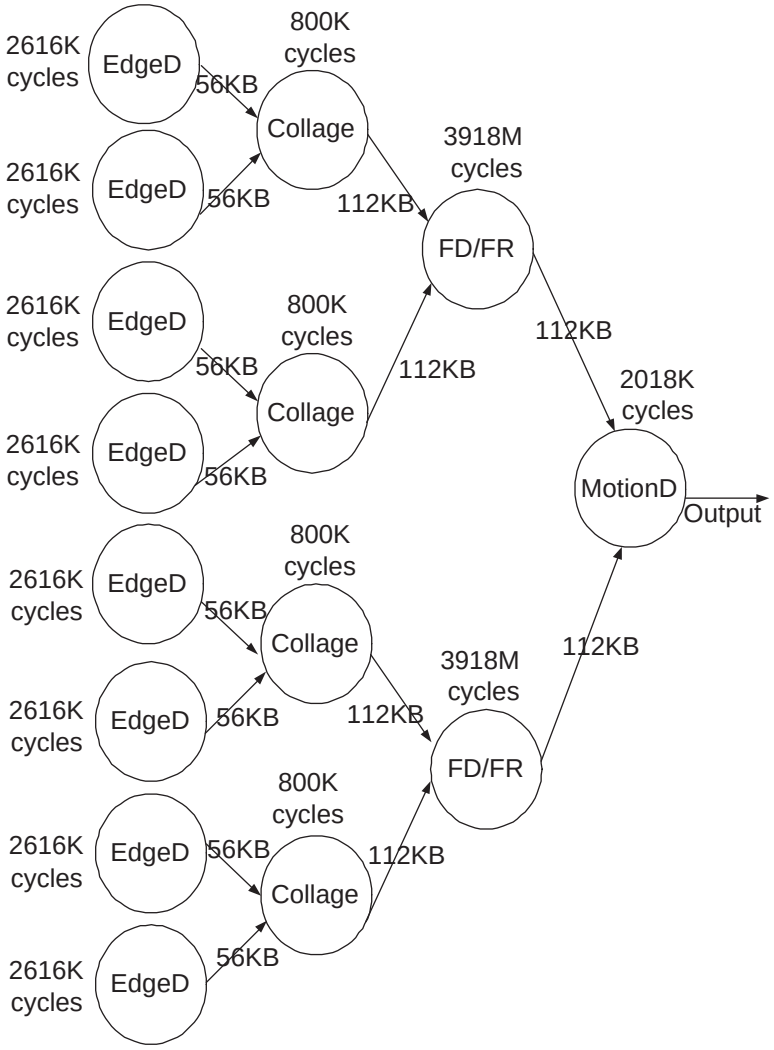


Figure 40: Video-based Tracking Dataflow Graph

I measure the average time taken per output data item for the 15 stage (8 *EdgeD*, 4 *Collage*, 2 *FD/FR*, 1 *MotionD*) dataflow graph with the three different algorithms under four different resource configurations picked randomly between the various resource configurations experimented with earlier. The goal of the study is to evaluate the application performance under different schedules obtained by the periodic Streamline and distributed scheduling algorithm as resource configuration is changed. The experiments are performed on 15 nodes (1 processor in each node) of a cluster.

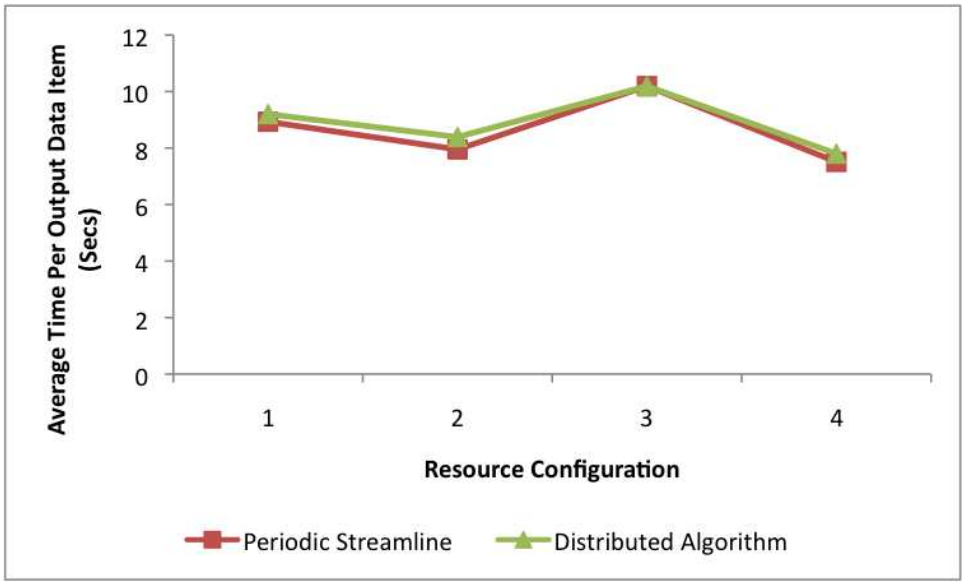


Figure 41: Scalability of Different Scheduling Algorithms

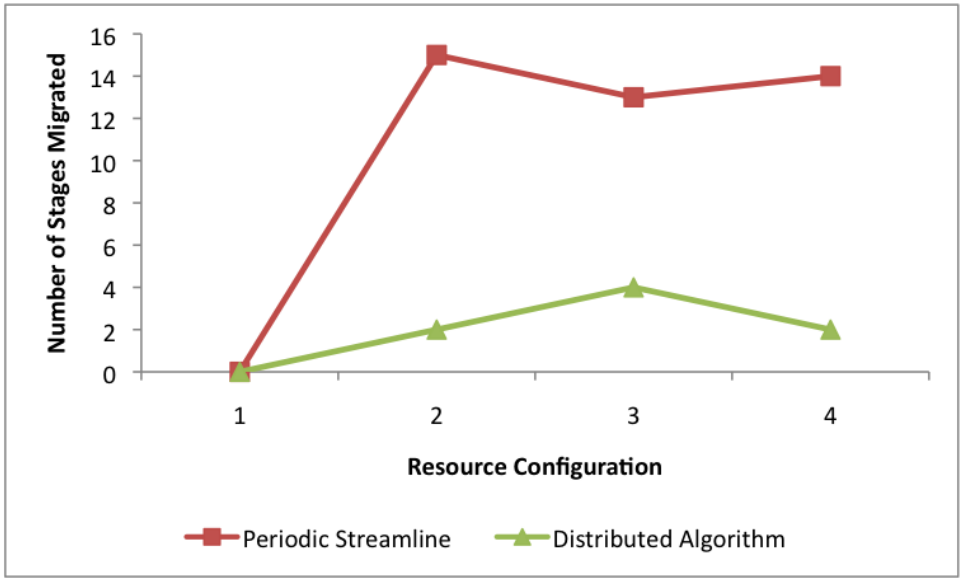


Figure 42: Overhead of Different Scheduling Algorithms

Figure 41 shows the performance of periodic Streamline and the distributed algorithm under different resource configurations. As can be seen from the figure, the distributed scheduling algorithm performs close to (within 5%) the periodic Streamline algorithm under all resource configurations.

Figure 42 shows the overhead (as measured by number of stages migrated) of the periodic Streamline and the distributed scheduling algorithms. As can be seen from the graph, periodic Streamline results in very high overhead requiring the complete dataflow graph to be re-instantiated in certain cases (as resource configuration changes from one to two and again from three to four). On the other hand, the distributed algorithm is able to deliver acceptable performance with much less overhead (7.5 times less overhead requiring only two stages to be migrated instead of fifteen).

6.6 Related Work

Scheduling applications on multiprocessors has been studied extensively in the literature. Braun et al.[15] give an overview of different scheduling heuristics in this space. These scheduling algorithms cannot be directly applied to streaming application dataflow graphs primarily because streaming applications are continuous in nature and the complete task graph needs to be scheduled (taking into account both computation and communication requirements) to deliver the desired quality of service.

Scheduling directed acyclic graphs (DAG) has also been studied extensively in the literature in the context of homogeneous multiprocessor systems as well as in the context of heterogeneous platforms. Deelman et al.[30] provide an overview of scientific workflow systems and their capabilities. Traditional researches into scheduling workflow for grid applications have focused on optimizing the time difference between the start and finish of the sequence of tasks. The most popular approaches are the static or dynamic list scheduling algorithms based on different heuristics, such as opportunistic

load balancing (OLB), minimum execution time (MET), minimum completion time (MCT) [15], random [56], and heterogeneous earliest finish time (HEFT) [80]. The basic idea in these different approaches is to determine an order of tasks based on heuristics and schedule the tasks according to the order. Besides, there are also some meta-heuristic approaches, such as genetic algorithms (GAs) [84, 58], and simulated annealing (SA) [50]. All these algorithms focus on determining the order of the tasks in the workflow because one task needs to complete before subsequent tasks in the workflow can be started. A streaming application dataflow graph differs from these workflows because the application needs to be continuously running. Therefore, the dynamic scheduling problem for a streaming application dataflow graph needs to continuously adapt the scheduling decision for each stage of the dataflow graph based on observed quality of service of the application.

SODA [87] has design goals somewhat similar to us. It is aimed at handling complex jobs involving enormous amount of streaming data. SODA allocates resources to the admitted jobs and controls job submission. However, SODA at this point focuses on one time placement as jobs are submitted to the system whereas our focus here is dynamic adaptation of the scheduling as resource availability changes.

In [90], authors formulate the streaming application mapping problem as a modified version of graph isomorphism and present an algorithm for static scheduling. They compare the performance of their approach with Streamline algorithm and show that it performs 30% better on average by optimizing for the most bottleneck path in the dataflow graph. However, they focus primarily on static scheduling whereas I have presented a more comprehensive framework for data streaming applications.

6.7 Contributions

I make the following contributions in our dynamic scheduling work: (*i*) I present a system architecture for dynamic scheduling of streaming applications; (*ii*) I define application *entry-points* for seamless migration and dynamic scheduling; (*iii*) I compare two dynamic scheduling algorithms (periodic Streamline and distributed scheduling algorithm) and demonstrate that the distributed algorithm is able to deliver comparable performance (within 5%) to periodic Streamline algorithm with much less overhead (7.5x less overhead in the scalability study).

CHAPTER VII

UBIQSTACK: A TAXONOMY FOR A UBIQUITOUS COMPUTING SOFTWARE STACK

7.1 Motivation

In prior chapters, I presented a uniform programming abstraction as well as static and dynamic scheduling algorithms that make it easier to develop streaming applications. Data streaming applications fall within the class of ubiquitous computing applications because they utilize the ambient computing infrastructure. As a case study of such data streaming and ubiquitous applications, I contributed to the development of a software intercom that allows users throughout a home to communicate through walls and floors as if they were in the same room. Using the experience gained during the system development and by analyzing existing technologies at the time, we came up with a taxonomy of ubiquitous computing software stack called UbiqStack[61]. Through the lens of the UbiqStack taxonomy, we survey a variety of subsystems designed to be the building blocks from which sophisticated infrastructures for ubiquitous computing are assembled. Our experience shows that many of these building blocks fit neatly into one of five UbiqStack categories, each containing functionally-equivalent components. In this chapter, I describe the UbiqStack classes of subsystems and explore each in light of the experience gained over two years of active development of both ubiquitous computing applications and software infrastructures for their deployment.

A ubiquitous computing (UbiComp) environment entails (*i*) an extensive and complex computer architecture deployment, (*ii*) sophisticated conduits for dynamic data

flow and control across this architecture, and (*iii*) a judicious external interface facilitating user interaction with the system. A ubiquitous infrastructure acts very much like a telephone patch board i.e., a software layer which various assets are "hooked" into and where logical and physical connections between these assets are instantiated. Assets, in the broadest sense, refer to physical devices, bandwidth, processors, as well as services made available through the patch board. This analogy suggests a standard and uniform, but not necessarily central, interface. It is only through this middleware layer that the vast array of disparate assets can be brought together to create the type of rich ubiquitous applications we are excited about.

A necessary precondition for deployment of a UbiComp application is a distributed system composed of (*i*) computational resources including network connected sensors and actuators, and (*ii*) a mechanism to generalize patterns of interaction with these resources. The reusable corpora of middleware can speed the second step, but in our own experience,we have often over-extended or misused existing UbiComp middleware subsystems. A standard language for describing middleware subsystems or a simple taxonomy for classifying subsystems could aid in the identification of potential candidates for a middleware deployment. Different distributed applications, each using middleware for intra-application communication, have very diverse needs. Similarly, an individual distributed application might, at times, require very different things of middleware subsystems. In these cases, multiple, complementary middleware subsystems might be used simultaneously across a single computer architecture deployment to provide the most effective communication tools to all applications. Here the middleware taxonomy must also provide aid in choosing what multiple middleware subsystems might be deployed simultaneously to the greatest effect. To arrive at a reasonable taxonomy describing UbiComp infrastructure and to understand the possible infrastructure interactions, we begin by examining some current and potential UbiComp infrastructures such as UPnP, MediaBroker,and the GRID. By partitioning

these varied middleware subsystems into categories/ equivalence classes, it becomes possible to reason about entire infrastructures piecemeal. Thus a whole infrastructure may be evaluated in terms of its constituent parts, streamlining the evaluation process and the infrastructure's subsequent refinement. This evaluation allows isolation of individual infrastructure system faults as well as isolation of security concerns.

In addition, standard subsystem classes can improve development of standard infrastructure, system-to-system interaction patterns. With a clearer picture of what infrastructure capabilities exist and where they are located, developers are better positioned to leverage these capabilities in their applications. Once the application requirements have been identified, developers may be able to pick existing infrastructure pieces to develop the application quickly and effectively.

This chapter proposes UbiqStack: a five-class infrastructure taxonomy based on the orthogonal functionalities of most commonly occurring subsystems. These include (*i*) Registration and Discovery, (*ii*) Service and Subscription, (*iii*) Data Storage and Streaming, (*iv*) Computation Sharing and (*v*) Context Management.

Figure 43 shows a visualization of our UbiqStack corresponding to top to bottom interaction between the equivalence classes. The user application is provided with the APIs for interacting with Services, Streams and Storage Locations, and Process Migration/ Distribution. Context Management Services and Ontologies are also common among all applications. This chapter continues in Section 7.2 discussing the imagined computer architecture deployment for UbiComp as well as some initial application deployment experience using middleware to bridge between application and hardware deployment. Section 7.3 details the taxonomy classes of UbiqStack. Section 7.4 discusses securing a ubiquitous computing infrastructure in UbiqStack terms. Finally, I conclude in Section 7.5.

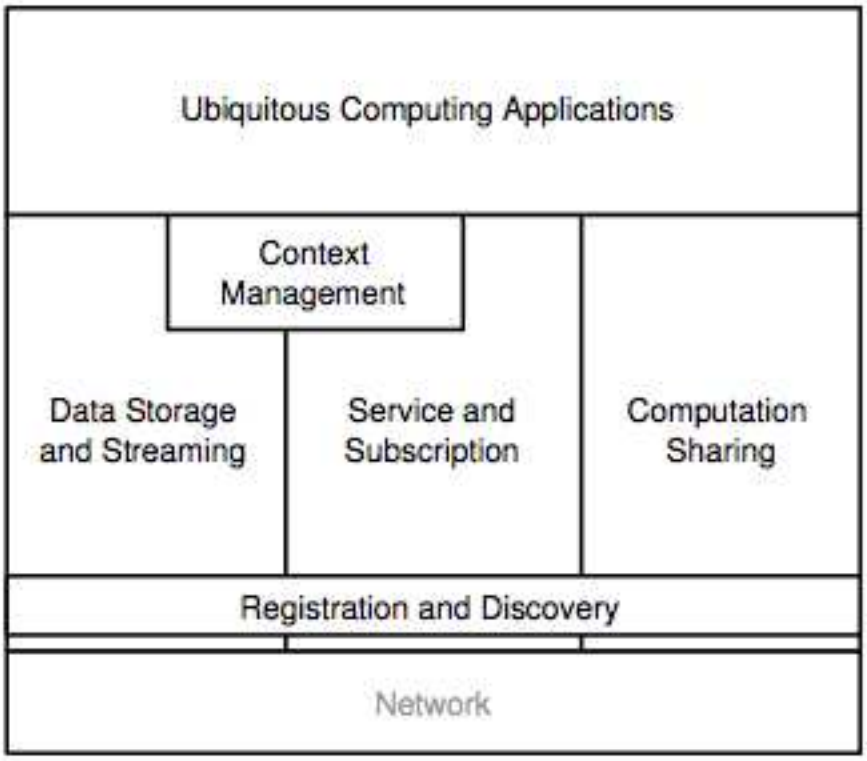


Figure 43: UbiComp Infrastructure Component Classes

7.2 Examining Existing Infrastructure Components

The intrinsic nature of common UbiComp applications builds on connecting distributed nodes into a over-arching application. This application might distribute itself onto multiple computers as a common Internet messaging client is deployed on multiple computers throughout the world, while each individual client connects into a larger whole that is a global messaging application. An application might also run in a single location but request and access resources available throughout its environment. As an example, a UbiComp television set-top box might request both a program stream from the cable company and display permission on an autonomous television display. The set-top box could then provide advanced television watching functionality as it routed the stream from the cable company to the television.

A common infrastructure built to support UbiComp applications must then provide the ability for applications to use computation and data resources throughout the environment. Previously we discussed UbiComp infrastructure as the combination of deployed computer hardware and the middleware stitching it together. Before we discuss the middleware subsystems running across the hardware, we will discuss the hardware deployment paradigm we are working from and our previous experiences in building applications on top of hardware deployed according to this paradigm.

7.2.1 A Deployment Paradigm

Striving toward the deployment of UbiComp applications in smart spaces, we have been developing both applications and infrastructures with a simple paradigm in mind. This paradigm is constructed of simple, modular, moderately capable computing devices being connected directly to sensors or sinks throughout a space. In this paradigm each computer can proxy access to its directly connected devices over its network interfaces. Transient devices and stationary devices are connected together abstracting over connection media to form connected graphs of computational devices. Each device then is a node in the graph.

Each of these distributed nodes will provide simple building block resources that applications might query and use. This way, an application might require a camera stream or a speaker, query for devices containing that service in the area requested, then request and use the resource. Each of these resources as well as the infrastructures exposing these resources to the outside world are managed as modular building blocks.

7.2.2 Development Case Study

Our software intercom is an example of an application built on existing hardware and middleware, deployed with the above paradigm in mind, in the Aware Home at Georgia Tech[49]. Using consumer electronics computers, microphones, speakers, and our

middleware, this software intercom allows users throughout a home to communicate through walls and floors as if they were in the same room.

In accordance with the paradigm described above, applications expose resources which other applications use. The user interacts with an intercom interface application which controls a set of modular audio applications deployed throughout the Aware Home. The user interface application projects a control-information resource which outputs current state information while the audio applications provide audio resources which stream captured audio. While running, The interface application queries the state of the audio resources and actuates end-to-end connections through its single control resource. In turn each audio application listens to the interface application's control resource for direction.

Our initial deployment relied on one-way data streams to transfer audio information between audio inputs and outputs between audio access applications while using the same one-way data streams to distribute control information from the interface application to the audio applications. The nature of these one-way, relatively static, connections between controller and controlled made the deployment of the intercom much less dynamic than anticipated.

The intercom has been redesigned to use MediaBroker[62] as an infrastructure for low latency audio data streams and UPnP for distributing discrete control messages between a set of control applications and the audio applications they control. In this case, taking advantage of UPnP for low-bandwidth message passing allows us to apply MediaBroker to the problem it better solves - moving audio data from sources to sinks according to data requests while offloading control message handling to UPnP. UPnP has better facilities for creating the interface application's control resource and for allowing the many audio access applications to better listen to multiple interface applications at once.

Using the strength of each subsystem and the scaffolding each offers, the job of

programming the audio application and the control application is greatly simplified. Using the standard interfaces provided by UPnP and MediaBroker cleanly, the programmer can concentrate on the individual concerns of each application and not how to translate those concerns into something the middleware can do. With a well-defined taxonomy with which to understand UbiComp middleware subsystems, implementation of this intercom would begin with brief modeling of all intra-application and inter-application interactions based on the capabilities of each middleware subsystem categories' abilities.

7.3 Subsystem Category Overview

In the brief case study above, system deployment and application development were made simpler through leveraging an existing technology in the context of a novel problem.This anecdotal evidence is in fact exemplary of the benefits derived through clean separation of subsystems. As we have seen, the delineation and choosing of subsystems along functional boundaries enables us to better reason about the infrastructure as a whole as well to improve the quality of a specific application. Each of these existing subsystems filled a specific role in the development of the application and in the larger infrastructure deployment.

Similar in functionality to UPnP[47], most WebServices implementations provide the ability to make remote calls to procedure located in other applications on other hosts. We argue, then, that UPnP and WebServices provide interchangeable functionality, which means that they belong to the same category of subsystems, namely Service and Subscription, and might be fairly compared to each other when selecting the best-fit subsystem for the job at hand. In turn, their current limitations may serve as a driver for development of a more robust replacement. In turn, the high-bandwidth, low-latency streaming facilities offered by MediaBroker is what earns it its place in

a different category of subsystems, namely Data Storage and Streaming. Other subsystems exist to accomplish similar tasks: GnuStream [48] and D-Stampede [7] while still other subsystems provide shared, network accessible, data stores: Coda [74] and T-Spaces [54].

7.3.1 Registration and Discovery

At the heart of UbiqStack, registration and discovery facilitates connection of all other UbiqStack components. Communication between applications running on top of the distributed nodes of a UbiComp infrastructure begins with discovery of available resources. Once different applications connect, they can share descriptions of their capabilities and begin to collaborate on higher order tasks. The Semantics of discovering distributed resources (whatever they might be) varies along several independent dimensions. One is whether the resource is actively published/advertised, the other is where the resource is marked globally accessible. Another dimension is the level of flexibility in querying.

Myriad languages, protocols, and frameworks exist for the discovery of hosts on the network, applications running on hosts, and services and resources provided by those applications. Common to the WebServices and UPnP world are protocols like SSDP [38], UDDI [31], and WSDL [41]. These allow robust and systematic description and discovery of services. Jini provides significant functionality in the area of service discovery as well as actual resource migration once it is discovered. Many lightweight services like DNS and LDAP provide a more static registration mechanism suitable for resources with considerably longer lifetimes. Specialized facilities like BSDP, which is tied to Bluetooth protocol stack, are also popular.

Evaluating registration and discovery subsystems based on their capabilities must take into account efficiency and extensibility. Some subsystems will require strict and complex registration as well as manual discovery and iterative search of what is

available. Other subsystems will provide functionality to automate much of this.

7.3.2 Service and Subscription

Service and subscription, the primary pillar of UbiqStack, encompasses the passing of discrete messages to accomplish distributed service registration and access. For example, an application might offer a resource capable of converting US zip codes to map images. Applications across the network can then access this resource to accomplish this. Applications might also want to know when some event happens and they might subscribe to be notified of some change. Both service and subscription rely on discrete messages being reliably passed between applications.

According to the paradigm discussed above, UbiComp applications will consist of many smaller parts that will require intra-application communication for synchronization. The discrete information exchange provided by a service and subscription subsystem allows distributed application entities to keep state between themselves and provides the necessary abstractions for setting state on adjacent nodes and being updated with their states when other nodes' states change.

The service and subscription component must provide an extensible application programming interface. The component must also be able to scale with respect to the number of applications using the subsystem and the size of data being transferred between applications. The speed with which remote services are called and with which subscription updates are propagated is also integral to the evaluation.

A service and subscription subsystem we have discussed previously in this chapter, UPnP, is really only mediocre at message passing and has limited functionality allowing extensibility. UPnP's scalability is also limited to a single network subnet without resorting to complex bridging technology. Context Toolkit [73] provides a service and subscription component as well as some basic context reasoning ability but it has many of the same limitations of UPnP in its inefficiency. Webservice

implementations vary greatly, but many provide very efficient messaging and fairly extensible application programming interfaces.

7.3.3 Data Storage and Streaming

Beyond the simple message passing described above, many UbiComp applications, especially those geared toward multimedia capture and access, require significant, ordered data transfer. The second UbiqStack pillar, data storage and streaming, allows distributed applications to share large amounts of data in a structured manner. For multimedia streaming, structure might emerge in the ordering of video frame delivery, while other applications might require the structure necessary to address, request and change a single value in very large data structure. A data storage and streaming component must be capable of handling significant structured data movement between distributed nodes. This data movement might involve streaming multimedia from a media provider to a media consumer or it might encompass more complex requirements for data archival and retrieval.

Streaming middleware subsystems like MediaBroker [62], GnuStream [48], Indiva [66] and D-Stampede [7] allow distributed nodes to ship streams of data to each other, while other subsystems like Server-less File Systems [10] and Coda [74] provide distributed access to common filesystems. Infrastructures like T-Spaces [54] or JavaSpaces [72] provide more generic sharable network memory. All of these classes provide access to large amounts of data in a structured manner. DFuse APIs, described in earlier chapter, also fall within this class of data storage and streaming.

Measuring the performance of a data storage and streaming subsystem might be difficult if we only take into account efficiency at providing quality of service. Many data storage and streaming subsystems provide varied types of functionality including different delivery or data set management schemes.

7.3.4 Computation Sharing

The third UbiqStack pillar, Computation sharing allows a running application to make use of remote computational resources on demand. For example, a surveillance application running on a sensor network may decide to offload complicated vision tracking components onto more capable high performance computing resources (HPC) strategically placed throughout the ubiquitous infrastructure. Furthermore, in order to accommodate mobile clients to deliver required quality of service, such computation must be allowed to move from one HPC resource to another. In order to support such unfettered, yet highly-specialized access to computing cycles, a mechanism for discovering unclaimed CPU time and dynamically scheduling computations on the HPC resources is essential.

Many tools for enabling distribution of compute-intensive tasks have emerged over the years. The scientific computing community is largely responsible for this variety. Regrettably, the focus has often been on batch processing, which alone is insufficient to support rich ubiquitous computing applications we envision. Grid computing allows controlled sharing of computation resources across many different physical organizations in order to deliver a quality of service. Open Grid Service Infrastructure(OGSI)[33] has extended web services by providing standard mechanism for creating, naming and discovering grid services, providing virtualization through location transparency, and supporting integration with underlying native platform facilities. The Globus Toolkit [34] built on OGSI model, provides the middleware support needed to build grid computing applications. MPI [32], PVM [36] are some of the other tools that can be used for resource sharing within set of connected computers. These tools provide libraries for building scientific computing applications and running them on multiple computers at once. However, the capabilities of these tools don't extend very well for streaming applications. Moreover, they don't provide capabilities for dynamically asking additional resources or dynamic joining of

computational entities.

Facilities for computation sharing must be dynamic enough to provide for application, thread or even function migration across multiple distributed entities. Applications might want to dynamically move through the environment in order to physically follow a user. Applications might also simply want to spread computationally intensive processes to more capable elements in the environment. The tools for computation sharing should also support dynamism in terms of the resources being allocated, the computation being performed as well as the mobility of the user. The Streamline heuristic and the dynamic scheduling algorithms presented in earlier chapters address these aspects.

7.3.5 Context Management

A context management subsystem defines a common language to describe the context of an environment and also defines a language to discuss and describe the context state. This middleware subsystem is built on both the service and subscription components as well as the data sharing and streaming components as it will commonly differentiate streaming sensor data into discrete values. While this differentiation will happen at the ubiquitous application level, the differentiation process can be represented inside the context state.

Much of UbiComp relies on having access to a consistent view of the world so that applications might correctly actuate themselves according to the environment and those in the environment. This common context management subsystem component allows applications to keep current on the state of the world by communicating with each other in a language common to all possible applications in the space.

Important in the realm of context management, synonym and unique naming management have been studied fairly thoroughly by multiple projects under the Semantic Web [17] umbrella and also by projects like OntoWeb [77] and Context ToolKit [73].

7.4 Compartmentalizing Security with UbiqStack

The difficulty in deploying ubiquitous computing infrastructure components to enable application level communication is compounded by the need to provide secure apparatus for this communication. Envisioned ubiquitous computing applications require capture, transmission and analysis of extremely sensitive information. With video cameras capturing our every move, large data sets describing what each of us is doing, and computers ready and willing to help less capable computing devices with calculations, system security becomes paramount. Following from the traditional "CIA model" [67] of ensuring Confidentiality, Integrity and Availability, a secure ubiquitous computing infrastructure would demand that all communication remained secure and authentic.

Similar to how the UbiqStack taxonomy allows description of complex infrastructures in terms of components, we believe security is best described as a per component responsibility. While an orthogonal set of security policy standards will enable each infrastructure component to consult access control lists or key servers in a standard way, each infrastructure component requires unique security procedures which an over-arching security layer cannot handle. Each infrastructure component will implement a common security policy in some verifiable way that is meaningful to that component, but such that the composition of all the components will still retain the properties of the model and access control policy.

7.4.1 Questioning Security

The UbiqStack component categories each require that specific security concerns be addressed. Here we examine hypothetical security questions and analyze why these questions are unique to each of the UbiqStack categories.

7.4.1.1 Registration and Discovery: Can I Know If a Service or Resource Exists?

An infrastructure component providing secure registration and discovery scaffolding must address the very specific needs of applications attempting to register and later discover a service or resource online. Often simply acknowledging that you are interested in knowing something can reveal details best kept private: "Does any-one know where I can refill my prescription for heart medication?"

7.4.1.2 Service and Subscription: Can I Call This Service?

Where the registration and discovery infrastructure component had the specific duty to ensure only legitimate parties could communicate, the service and subscription infrastructure component must protect all communications related to the discrete message interchange of remote service calls and subscription delivery. Beyond securing the messages passed between applications, a service and subscription component must ensure that all messages are legitimate and from legitimate entities.

7.4.1.3 Data Storage and Streaming: Can I See This Stream or Access this Store?

While similar to the question for the service and subscription component, it is inherently different because of the semantics of the data being traded/addressed. Where a service might be able to authenticate or exchange keys for each message traded back and forth, the sheer volume traded by data streaming and storage infrastructure components require different access rights management. The data volume also mean that multiple consumers will probably be consuming the same data where the service and subscription data can be encrypted and tailored specifically for each caller.

7.4.1.4 Computation Sharing: Can I Run This Code Here?

Establishing trust between applications running a computation sharing infrastructure component is possibly the most critical in terms of security for ubiquitous computing. Here alone in the UbiqStack taxonomy is actual, possibly destructive, logic being

moved between applications. Secure computation sharing components must ensure that logic is verified for safety, and appropriately contained to prevent malicious code from causing destruction.

7.4.1.5 Context Management: Can I Know Where Someone Is?

The context management infrastructure component must provide security that is no longer specifically technical. With the UbiqStack concepts in mind, we expect an application to consult a secure context management resource only after finding it with a secure registration and discovery component and establishing a connection over a secure service and subscription infrastructure component. Here a technically secure connection has been established between querier and queried and what remains to be established is a socially secure connection. The context management UbiqStack component must again use common access control lists and public key server verify that personal and social facts are accessible.

7.4.2 Building Security

This discussion of security relies on the separation of each infrastructure component's security concerns. Allowing each component to manage its own security will lead towards portable, extensible systems. The redundancy of security might also allow for better technical security. In a similar manner, modern wireless and web technologies use compartmentalized security to establish secure wireless communication (via WEP or other) before establishing a secure application layer SSL channel to perform what would finally be called a secure credit card transaction [89]. Here the SSL communication channel secures the application layer communication while the WEP encryption secures network packet transmission. The two security apparati are unaware of each other, but as each does its best to secure the communication, the system is more resilient.

With UbiqStack component category security, we first expect a device to establish

some secure network connection. With network connection in place, the device would proceed to authenticate to registration and discovery peers where self-registration and peer discovery would occur. After discovering an interesting multimedia stream, the device would authenticate to the owner/carrier of the multimedia stream before the stream would be accessible. Through compartmentalizing security within each UbiqStack component, secure applications can be built on secure infrastructure components.

7.5 UbiqStack Conclusions

In this chapter, we have presented a comprehensive UbiComp middleware taxonomy, UbiqStack, to ease discussion and facilitate the process of building both ubiquitous middleware subsystems as well as ubiquitous applications. We also discussed where in UbiqStack existing middleware subsystems fit and how security questions are best addressed by each individual subsystem. In our experience, the sampling of industrial and academic experience and the combination of knowledge across groups allows better middleware construction and UbiqStack hopes to further these interactions through the creation of a common language for middleware discussion. We are not entirely sure about the feasibility of creating complex infrastructures by picking and choosing subsystems from each taxonomy category, but we look to examine it. Much research still to be done at the intersections of these UbiqStack categories to explain the gaps that exist and also to examine their interactions. Research can also be directed to survey entire UbiqStack middleware categories; learning from the state of the art and designing and building new class-complete infrastructures.

The focus of this thesis is primarily addressing Data Storage and Streaming, and Computation Sharing component in UbiqStack. DFuse provides common abstractions that facilitate data storage and streaming while hiding the complexities of distributed environment. Streamline and dynamic scheduling algorithms facilitate access

to shared resources and enable computation sharing.

CHAPTER VIII

CONCLUSION

Developing Streaming applications is a challenging task because (*i*) These applications require access to heterogeneous resources from limited capability sensors to computationally powerful HPC resources; (*ii*) The applications have real-time quality of service requirements (such as bounds on latency and throughput); and (*iii*) the applications are distributed and present various issues (such as failure, buffer management, and synchronization) that are inherent in distributed programming. I propose that:

An intuitive programming abstraction will make it easier to build *dynamic*, *distributed*, and *ubiquitous* data streaming applications. Moreover, such an abstraction will enable an efficient allocation of *shared* and *heterogeneous* computational resources thereby making it easier for domain experts to build these applications.

This dissertation investigates novel middleware mechanisms in support of the above thesis. The contribution of this work is five fold.

First, I present a programming abstraction called, DFuse, that makes it easier to develop these fusion applications. DFuse embodies a detailed API for supporting fusion application processing. The fusion API in the DFuse architecture subsumes issues such as data synchronization and buffer management. Through micro-benchmark of the primitives provided by the fusion API, I demonstrate that DFuse API implementation has acceptable overhead. API overhead is further reduced when we take into account various optimizations (such as pre-fetching, partial fusion, and data caching) that are implemented in DFuse. DFuse Implementation is used to evaluate various role assignment heuristics in sensor network.

Second, I address the problem of allocating high performance computing resources while delivering the required quality of service to the application. I present a scheduling heuristic, called Streamline, that takes into account following parameters in its scheduling decision: (a) computation and communication requirements of the various stages of the dataflow graph, (b) any application-specified constraints, and (c) current resource (processing and bandwidth) availability. The performance of Streamline is compared with Optimal placement, Simulated Annealing (SA) approximations, and E-Condor - a streaming grid scheduler that uses Condor. Through extensive experiments, I show that Streamline performs close (within 1%) to Optimal and Simulated Annealing, and is better than E-Condor by nearly an order of magnitude when there is non-uniform CPU resource availability, and by a factor of four when there is non-uniform bandwidth availability. Through scalability studies I show that Streamline is more effective than E-Condor in handling large dataflow graphs, and performs close to Simulated Annealing algorithms, with much smaller (by a factor of 1000) scheduling overhead. E-Condor is not aware of the computation and communication requirements of the dataflow graph and therefore performs much worse than Streamline. However, the results also demonstrate that Streamline's performance is close to Optimal for both small and large dataflow graphs.

Third, I implement Streamline as a grid service and evaluate it in wide area environment using Planetlab. I demonstrate that Streamline can be implemented as a grid service facilitating the allocation of resources and launch of streaming applications from a high level specification. Further, the experiments on Planetlab use real resource availability information by contacting various other grid services configured on each machine. The experimental results confirm that schedule generated by Streamline is close to optimal placement (proxied by Simulated Annealing algorithm) under real workload in a shared wide area environment.

Fourth, I propose a dynamic scheduling algorithm to address the limitations of

the one time placement generated by Streamline. I design and implement a system infrastructure that uses the Streamline algorithm periodically to adapt the placement of the dataflow graph nodes on available resources. The infrastructure consists of a programming primitive that facilitates dataflow graph migration, periodic evaluation of streamline algorithm, and an architecture for failure handling. Through experiments using a compute bound kernel, I show that the periodic evaluation of the Streamline heuristic results in 3.5x better performance when compared with one time placement using Streamline under dynamic resource conditions.

Finally, using a case study of such data streaming and ubiquitous applications, I come up with a taxonomy of ubiquitous computing software stack called UbiqStack. UbiqStack consists of five complementary functionalities most commonly needed in ubiquitous applications. These functionalities include *Registration and Discovery*, *Service and Subscription*, *Data Storage and Streaming*, *Computation Sharing* and *Context Management*. While DFuse provides a common API for data storage and streaming, Streamline and dynamic scheduling address the need of computation sharing for data streaming applications. Through the lens of the UbiqStack taxonomy, we survey a variety of subsystems designed to be the building blocks using which sophisticated infrastructures for ubiquitous computing are assembled. UbiqStack provides a common framework using which future middleware for data streaming applications can be compared.

In summary, the Fusion Channel programming abstraction makes it easier for domain experts to develop data streaming applications. An application only needs to specify the input and output connections to a channel, and the fusion function. The subsystems developed in this dissertation take care of instantiating an application, allocating resources for the application (via scheduling heuristics) and dynamically managing the resources (via dynamic scheduling). Through performance evaluation, I demonstrate that the resources are allocated efficiently to optimize the throughput

and latency constraints of an application. I have established the thesis through extensive micro measurements and scalability studies. In the next chapter I highlight some of the future work that remains as a follow up to this dissertation.

CHAPTER IX

FUTURE WORK

Several interesting directions for further research became apparent during the course of this work. This chapter gives an overview of such potential future research directions.

9.1 Programming Abstraction

A uniform programming abstraction makes it easier for domain experts to build streaming applications without worrying about the complexities of distributed programming. DFuse APIs take a step in that direction by providing the fusion channel abstraction. However, the fusion channel abstraction can be extended in following ways.

9.1.1 Multiple Streaming Applications

DFuse APIs and experiments have primarily focused on specification of a single dataflow graph. Because of many different use cases for video and audio stream analysis, multiple dataflow graphs may need to be instantiated and sustained on heterogeneous resources. For example, a video-based surveillance application may use a Face Detection fusion module to identify a suspicious person. A Social Computing application may use the same Face Detection fusion module to identify the location of a friend. The nodes of the dataflow graph for different applications may be shared to eliminate duplicate computations and to provide better quality of service to the applications.

It is possible to instantiate multiple data streaming applications using fusion channel abstraction. It is also possible to share computations and data across multiple applications using the fusion channel abstraction developed as part of my work. However, it puts the onus on the domain expert to know about all the applications currently available in the environment as well as leverage existing fusion channels wherever possible. It is difficult for a domain expert to have complete knowledge of the computing environment. Also it is very difficult, if not impossible, for a person to make an optimal decision across multiple applications. Moreover, the optimal decision would change as new fusion applications are instantiated or removed from the system. Therefore, there is an opportunity to extend the DFuse abstraction to make it easier to instantiate multiple data streaming applications. Some of the problems that need to be addressed are: (*i*) Discovering existing fusion channels in the system; (*ii*) eliminating duplicate computation and resulting duplication in data across multiple applications; (*iii*) providing abstractions that make it easier to deploy multiple streaming application dataflow graphs from higher level specifications; and (*iv*) dynamic adjustment to the fusion channels as applications are added or removed from the system.

9.1.2 Dynamic Dataflow Graph

The dataflow graphs of streaming applications may change at run-time with fusion channels being added or removed from an application. For example, in a distributed surveillance application, face detection computation may need to be performed based on the result of a motion detection module. An application dataflow graph is specified statically in this work. As a possible extension, we need programming primitives that would allow fusion application dataflow graphs to be dynamically configured based on result of the computations being performed. Moreover, result from a fusion function may feed into upstream nodes in the dataflow graph thereby forming feedback loops.

DFuse APIs allow input and output nodes of a fusion channel to be changed at run-time. However, a domain expert needs to know when to adjust the application dataflow graph and how to reconfigure the application. Specification and instantiation of such dynamic dataflow graphs are challenging. The programming primitives have to be easy so that a domain expert can use it. At the same time, it has to be flexible enough so that the dataflow graph can be automatically reconfigured based on higher level requirements of the applications. Handling dynamic dataflow graph specification and deployment are avenues for future research.

9.1.3 Abstractions for Domain Experts

DFuse exposes the abstraction of fusion channels to a domain expert. The domain expert will need to specify inputs/outputs to a fusion channel and the fusion function. In recent years, the MapReduce [29] framework has become popular for processing large amount of data in a cluster. In the MapReduce framework, applications specify the input/output data locations and supply map and reduce functions. A map function maps input {key, value} pairs to a set of intermediate {key, value} pairs. Reduce function reduces the set of intermediate values which share a key to a smaller set of values. MapReduce framework is being used for offline batch processing of large amounts of data.

Although MapReduce framework cannot be used directly for streaming applications dataflow graph, there is a need for simple and powerful abstractions for streaming application specification. Perhaps streaming application dataflow graph can be specified as a series of map and reduce functions that need to be executed in sequence on each data stream. The underlying system can still use the functionalities provided by fusion channels, and leverage optimizations such as pre-fetching and partial fusion. Exploring this area further is another avenue for research.

9.2 Extensions to Resource Management

As part of this work, I presented a static scheduling algorithm called Streamline as well as a Distributed Scheduling algorithm for resource management of a streaming application. These contributions can be extended as follows.

9.2.1 Computation and Data Priority

The scheduling algorithms presented in this work allocate resources to different stages of a dataflow graph based on their computation and communication requirements. However, in a streaming application, all computations may not have equal priority. For example, in a video-based surveillance application, data coming from a particular region may have higher priority than the data coming from a region with no movement. The scheduling algorithms need to take into account the priority of the data for efficient allocation of resources.

9.2.2 Multiple Streaming Applications

As mentioned earlier, multiple data streaming applications may share a fusion point in order to efficiently monitor an environment. It is challenging to automatically discover the state of the system, and make optimization decisions taking into account requirements from different applications. In this work, I have considered only a single streaming application dataflow graph at a time. Considering multiple application dataflow graphs in the scheduling decisions is an avenue for future work.

9.2.3 Dynamic Scheduling

The dynamic scheduling algorithms presented in Chapter 6 were evaluated using synthetic workloads and without considering the overheads of stage migration. Facilitating stage migration automatically in a real deployment is a challenging task. The scheduling algorithms need to be extended to take into account the overhead of stage migration.

9.2.4 Deploying a Large Scale Streaming Application

The work done as part of this dissertation needs to be used in a real-world large scale data streaming application. The real world deployment would provide additional challenges and insights as the system scales to support more applications and data sources. Building and evaluating such large scale applications (similar to the ones described in Chapter 2) are fruitful future research directions.

9.3 Using Sensor Network and HPC Resources

The resource management system built as part of this work addresses the need for HPC resource management for data streaming applications. However, developing these applications requires access to a broad spectrum of resources from sensor nodes to HPC resources. Cloud computing virtualizes the resources so that domain experts do not need to worry about resource allocation when developing their applications. However, cloud computing, as we know it today, does not include sensing nodes but only includes compute and storage nodes. For facilitating streaming application development, a Cloud provider needs to efficiently allocate resources that are heterogeneous along multiple dimensions such as (*i*) CPU, memory, and network capabilities; (*ii*) power consumption; (*iii*) form factor; (*iv*) network connectivity; and (*v*) sensing modalities. Developing an intelligent infrastructure that automatically manages such diverse resources across multiple data streaming applications is a promising area for future research.

REFERENCES

[1] "Britain is the World's Surveillance Leader." http://yro.slashdot.org/yro/04/09/02/1825243.shtml?tid=158\&tid=172, Sep 2004. Slashdot News.

[2] "Facebook Statistics." http://www.facebook.com/press/info.php?statistics, June 2010.

[3] "Georgia Navigator." http://www.georgia-navigator.com/cameras, June 2010.

[4] "Intelligent Transportation Systems." http://www.its.dot.gov, June 2010.

[5] Abadi, D. J., Carney, D., and et al., "Aurora: A new model and architecture for data stream management," *VLDB Journal*, vol. 12, pp. 120–139, August 2003.

[6] Adam, T., Chandy, K., and Dickson, J., "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, pp. 685–690, Dec 1974.

[7] Adhikari, S., Paul, A., and Ramachandran, U., "D-stampede: Distributed programming system for ubiquitous computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems(ICDCS)*, (Vienna), July 2002.

[8] Agarwalla, B., Ahmed, N., Hilley, D., and Ramachandran, U., "Streamline: A scheduling heuristic for streaming application on the grid," in *Thirteenth Annual Multimedia Computing and Networking Conference (MMCN'06)*, (San Jose, CA), Jan 2006.

[9] Agarwalla, B., Ahmed, N., Hilley, D., and Ramachandran, U., "Streamline: Scheduling streaming applications in a wide area environment," *Multimedia Systems (MMSJ)*, vol. 13, pp. 69–85, September 2007.

[10] Anderson, T., Dahlin, M., Neefe, J., Pat-terson, D., Roselli, D., and Wang, R., "Serverless network file systems," in *Proceedings of the 15th Symposium on Operating System Principles*, (Copper Mountain Resort, Colorado), pp. 109–126, December 1995.

[11] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M., "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[12] Ayyub, S. and Abramson, D., "Gridrod: a dynamic runtime scheduler for grid workflows," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, (New York, NY, USA), pp. 43–52, ACM, 2007.

[13] Balazinska, M., Balakrishnan, H., and Stonebraker, M., "Contract-based load management in federated distributed systems," in *1st Symposium on Networked Systems Design and Implementation (NSDI)*, (San Francisco, CA), March 2004.

[14] Boulis, A., Han, C. C., and Srivastava, M. B., "Design and implementation of a framework for programmable and efficient sensor networks," in *The First International Conference on Mobile Systems, Applications, and Services(MobiSys)*, (San Francisco,CA), 2003.

[15] Braun, T. D., Siegel, H. J., Beck, N., Boloni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D.,

and Freund, R. F., "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Parallel and Distributed Computing*, vol. 61, pp. 810–837, 2001.

[16] Buyya, R., Abramson, D., and Giddy, J., "Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid," *High Performance Computing (HPC) ASIA*, 2000.

[17] Calvanese, D., Giacomo, G. D., and Lenzerini, M., "A framework for ontology integration," in *The First Semantic Web Working Symposium*, pp. 303–316, 2001.

[18] Chandrasekaran, S. and et al., "TelegraphCQ: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD'03)*, 2003.

[19] Chapin, S. J., Katramatos, D., Karpovich, J., and Grimshaw, A. S., "The Legion resource management system," in *Job Scheduling Strategies for Parallel Processing*, pp. 162–178, Springer Verlag, 1999.

[20] Chen, L. and Agrawal, G., "Resource allocation in a middleware for streaming data," in *2nd Workshop on Middleware for Grid Computing (MGC'04)*, (Toronto, Canada), Oct 18 2004.

[21] Chen, L. and et al., "GATES: A grid-based middleware for processing distributed data streams," in *Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-13)*, June 2004.

[22] Cherniack, M., Balakrishnan, H., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S., "Scalable distributed stream processing," in *Proc. Conf. for Innovative Database Research (CIDR)*, 2003.

[23] Cherniack, M. and et al., "Scalable Distributed Stream Processing," in *First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, (Asilomar, CA), January 2003.

[24] Coffman, E., *Computer and Job-Shop Scheduling Theory.* New York: Wiley, 1976.

[25] Couvares, P., Kosar, T., Roy, A., Weber, J., and Wenger, K., "Workflow in Condor," *Workflows for e-Science*, January 2007.

[26] Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C., "Grid information services for distributed resource sharing," in *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001.

[27] Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W., and Tuecke, S., "A resource management architecture for metacomputing systems," in *IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 62–82, 1998.

[28] de Assuncao, M. D., di Costanzo, A., and Buyya, R., "Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, (New York, NY, USA), pp. 141–150, ACM, 2009.

[29] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[30] Deelman, E., Gannon, D., Shields, M., and Taylor, I., "Workflows and e-science: An overview of workflow system features and capabilities," *Future Generation Computer Systems*, vol. 25, pp. 528–540, May 2009.

[31] et al., B., "Uddi version 2.03 data structure reference," July 2002.

[32] Forum, M. P. I., "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, 1994.

[33] Foster, I., Kesselman, C., Nick, J., and Tuecke, S., "The physiology of the grid: An open grid services architecture for distributed systems integration," tech. rep., 2002.

[34] Foster, I., "Globus Toolkit Version 4: Software for Service-Oriented Systems," in *IFIP International Conference on Network and Parallel Computing*, pp. 2–13, Springer-Verlag LNCS 3779, 2005.

[35] Foster, I. and Kesselman, C., "Globus: A metacomputing infrastructure toolkit," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, pp. 115–128, Summer 1997.

[36] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, Mass.: MIT Press, 1994.

[37] Gerasoulis, A. and Yang, T., "A comparison of clustering heuristics for scheduling DAGs on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 276–291, Dec 1992.

[38] Goland, Y. Y., Cai, T., Leach, P., and Gu, Y., "IETF Internet Draft: Simple service discovery protocol/1.0," Oct 1999.

[39] Graham, R., Lawler, E., Lenstra, J., and Kan, A. R., "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Annals of Discrete Mathematics*, vol. 5, pp. 287–326, 1979.

[40] Gu, X., Nahrstedt, K., and Yu, B., "Spidernet: An integrated peer-to-peer service composition framework," in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, (Washington, DC, USA), pp. 110–119, IEEE Computer Society, 2004.

[41] Gudgin, M., Lewis, A., and Schlimmer, J., "W3C Working Draft: Web services description language (WSDL) version 2.0 part 2: Message exchange patterns," Mar 2004.

[42] Heidemann, J. S., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., and Ganesan, D., "Building efficient wireless sensor networks with low-level naming," in *Symposium on Operating Systems Principles*, pp. 146–159, 2001.

[43] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. E., and Pister, K. S. J., "System architecture directions for networked sensors," in *Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.

[44] Hu, T., "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, pp. 841–848, Nov 1961.

[45] Intanagonwiwat, C., Govindan, R., and Estrin, D., "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Mobile Computing and Networking*, pp. 56–67, 2000.

[46] Jae-Hwan Chang and Leandros Tassiulas, "Energy conserving routing in wireless ad-hoc networks," in *IEEE INFOCOM*, pp. 22–31, 2000.

[47] Jeronimo, M. and Weast, J., *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play.* Intel Press, 2003.

[48] Jiang, X., Dong, Y., Xu, D., and Bhargava, B., "Gnustream: a p2p media streaming prototype," in *Proceedings of IEEE Internation Conference on Multimedia and Expo*, Jul 2003.

[49] Kidd, C. D., Orr, R., Abowd, G. D., Atkeson, C. G., Essa, I. A., MacIntyre, B., Mynatt, E. D., Starner, T., and Newstetter, W., "The aware home: A living laboratory for ubiquitous computing research," in *Cooperative Buildings*, pp. 191–198, 1999.

[50] Kim, J.-K., Shivle, S., Siegel, H. J., Maciejewski, A. A., Braun, T. D., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R. B., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., and Yellampalli, S. S., "Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment," *J. Parallel Distrib. Comput.*, vol. 67, no. 2, pp. 154–169, 2007.

[51] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P., "Optimization by simulated annealing," *Science*, vol. 220, 4598, pp. 671–680, May 1983.

[52] Kumar, R., Wolenetz, M., Agarwalla, B., Shin, J., Hutto, P., Paul, A., and Ramachandran, U., "DFuse: A framework for distributed data fusion.," in *Proceedings of ACM SenSys 2003.*, (Los Angeles, CA, USA), Nov. 2003.

[53] Kwok, Y.-K. and Ahmad, I., "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506–521, 1996.

[54] Lehman, T. J., McLaughry, S. W., and Wycko, P., "T spaces: The next wave," in *Hawaii International Conference on System Sciences (HICSS'99)*, 1999.

[55] Liang, J. and Nahrstedt, K., "Service composition for advanced multimedia applications," in *12th Annual Multimedia Computing and Networking (MMCN 2005)*, 2005.

[56] Lopez, M. M., Heymann, E., and Senar, M. A., "Analysis of dynamic heuristics for workflow scheduling on grid systems," in *5th International Symposium on Parallel Distributed Computing (ISPDC'06)*, pp. 199–207, IEEE, July 2006.

[57] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W., "Tag: a tiny aggregation service for ad-hoc sensor networks," in *Operating System Design and Implementation(OSDI)*, (Boston,MA), Dec 2002.

[58] Martino, V. D. and Mililotti, M., "Scheduling in a grid computing environment using genetic algorithms," *Parallel and Distributed Processing Symposium, International*, vol. 2, p. 0235, 2002.

[59] Massie, M. L., Chun, B. N., and Culler, D. E., "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience," *Parallel Computing*, vol. 30, July 2004.

[60] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E., "Equations of state calculations by fast computing machines," *J. Chem. Phys.*, vol. 21, pp. 1087–1091, 1953.

[61] Modahl, M., Agarwalla, B., Saponas, T. S., Abowd, G., and Ramachandran, U., "UbiqStack: a taxonomy for a ubiquitous computing software stack," *Personal and Ubiquitous Computing Journal*, Aug 2005.

[62] Modahl, M., Bagrak, I., Wolenetz, M., Hutto, P., and Ramachandran, U., "Mediabroker: An architecture for pervasive computing," in *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications*, March 2004.

[63] Nabrzyski, J., Schopf, J. M., and Weglarz, J., *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, Sep 2003.

[64] Netperf, "The Public Netperf Homepage: *http://www.netperf.org/*," 2003.

[65] Nikhil, R. S., Ramachandran, U., Rehg, J. M., Halstead, R. H., Joerg, J. C. F., and Kontothanassis, L., "Stampede: A programming system for emerging scalable interactive multimedia applications," in *11th International Workshop on Languages and Compilers for Parallel Computing*, 1998.

[66] Ooi, W. T., Pletcher, P., and Rowe, L. A., "Indiva: Middleware for managing a distributed media environment." (BMRC Technical Note), 2002.

[67] Parker, D., *Computer security handbook*. Wiley, 2002.

[68] Peterson, L., Bavier, A., Fiuczynski, M., and Muir, S., "Experiences implementing planetlab," in *Operating System Design and Implementation (OSDI'06)*, November 2006.

[69] Ramachandran, U., Kumar, R., Wolenetz, M., Cooper, B., Agarwalla, B., Shin, J., Hutto, P., and Paul, A., "Dynamic data fusion for future sensor networks," *ACM Transactions on Sensor Networks*, 2006.

[70] Ramamoorthy, C., Chandy, K., and Gonzalez, M., "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. on Computers*, vol. C-21, pp. 137–146, Feb 1972.

[71] Rehg, J. M., Loughlin, M., and Waters, K., "Vison for a smart kiosk," in *Computer Vision and Pattern Recognition*, June 1997.

[72] S. Microsystems, "Javaspace specification," March 1998.

[73] Salber, D., Dey, A. K., and Abowd, G. D., "The context toolkit: Aiding the development of context-enabled applications," in *Proceedings of the 1999 Conference on Human Factors in Computing Systems CHI'99*, pp. 434–441, 1999.

[74] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C., "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[75] Sih, G. and Lee, E., "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 75–187, Feb 1993.

[76] Singh, S., Woo, M., and Raghavendra, C. S., "Power-aware routing in mobile ad hoc networks," in *Mobile Computing and Networking*, pp. 181–190, 1998.

[77] Spyns, P., Oberle, D., Volz, R., Zheng, J., Jarrar, M., Sure, Y., Studer, R., and Meersman, R., "Ontoweb - a semantic web community portal," in *Fourth International Conference on Practical Aspects of Knowledge Management (PAKM)*, Dec. 2002.

[78] Talwar, V., Basu, S., and Kumar, R., "An environment for enabling Interactive Grids," in *12th International Symposium on High-Performance Distributed Computing (HPDC'03)*, (Seattle, WA), pp. 184–193, June 2003.

[79] Thain, D., Tannenbaum, T., and Livny, M., "Distributed computing in practice: the Condor experience," *Concurrency - Practice and Experienc*, vol. 17, no. 2-4, pp. 323–356, 2005.

[80] Topcuoglu, H., Hariri, S., and Wu, M. Y., "Performance-effective and low complexity task scheduling for heterogeneous computing," *IEEE Transanction on Parallel Distributed System*, vol. 13, no. 3, pp. 260–274, 2002.

[81] Topcuoglu, H., Hariri, S., and Min-YouWu, "Task scheduling algorithms for heterogeneous processors," in *Proceedings of the 8th Heterogeneous Computing Workshop*, pp. 3–14, 1999.

[82] Topcuoglu, H., Hariri, S., and Wu, M.-Y., "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, March 2002.

[83] Viola, P. and Jones, M., "Rapid object detection using a boosted cascade of simple features," in *Proc. CVPR*, pp. 511–518, 2001.

[84] Wang, L., Siegel, H. J., Roychowdhury, V. R., and Maciejewski, A. A., "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *J. Parallel Distrib. Comput.*, vol. 47, no. 1, pp. 8–22, 1997.

[85] Weiss, A., "Computing in the clouds," *netWorker*, vol. 11, pp. 16–25, December 2007.

[86] Wolenetz, M., Kumar, R., Shin, J., and Ramachandran, U., "Middleware guidelines for future sensor networks," in *First Workshop on Broadband Advanced Sensor Networks (BASENETS'04)*, (San Jose, CA), Oct 2004.

[87] Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., and Fleischer, L., "Soda: an optimizing scheduler for large-scale stream-based distributed computer systems," in *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, (New York, NY, USA), pp. 306–325, Springer-Verlag New York, Inc., 2008.

[88] Wolski, R., "Dynamically forecasting network performance using the network weather service," *Journal of Cluster Computing*, vol. 1, pp. 119–132, January 1998.

[89] Zhou, L. and Haas, Z., "Securing ad hoc networks," *IEEE network*, vol. 13, pp. 24–30, 1999.

[90] Zhu, Q. and Agrawal, G., "Resource allocation for distributed streaming applications," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, (Washington, DC, USA), pp. 414–421, IEEE Computer Society, 2008.